TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

EERO OINAS

TUOTANNONSUUNNITTELUJÄRJESTELMÄN SUORITUSKYKYMONITOROINTI

Diplomityö

Tarkastaja: professori Mikko Tiusanen

Tarkastaja ja aihe hyväksytty
Sähkö- ja tietotekniikan tiedekunnan kokouksessa 3. kesäkuuta 2015

TIIVISTELMÄ

EERO OINAS: Tuotannonsuunnittelujärjestelmän suorituskykymonitorointi
Tampereen teknillinen yliopisto
Diplomityö, 45 sivua
Joulukuu 2015
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: professori Mikko Tiusanen
Avainsanat: tuotantoympäristö, instrumentointi, Application Insights, aspektiohjelmointi

Työpöytäsovellusten suorituskyvyn mittaaminen lopullisessa käyttöympäristössä antaa kehittäjälle tietoa ohjelmiston laadusta ja käytettävyydestä varsinaisissa käyttötilanteissa. Suorituskyvyn mittaaminen kehitysympäristön ulkopuolella vaatii kuitenkin runsaasti kehittäjien resursseja, koska tiedonkeruutoimintojen lisääminen vaatii usein merkittäviä muutoksia valmiiseen koodipohjaan. Tässä työssä kehitetään toimivampi ratkaisu suorituskyvyn mittaamiseen käyttöympäristössä.

SW-Development on ohjelmistoyritys, joka kehittää pääasiallisesti tuotannonsuunnittelujärjestelmiä valmistavan teollisuuden tarpeisiin. Yrityksen tuotannonsuunnittelujärjestelmä, SWD Planning Effieciency System (SWD PES), tarjoaa asiakkaille tuotannonsuunnitteluun ja -optimointiin kehitettyjä työkaluja. Järjestelmä käyttää paljon tietokantadataa ja suorittaa raskaitakin laskentaoperaatioita. Järjestelmää kehitetään edelleen, ja siksi suorituskyvyn mittaaminen myös kehitysympäristön ulkopuolella on tärkeä osa jatkuvaa kehitystä.

Tässä työssä suunniteltiin ja toteutettiin komponentti suorituskykyinstrumentointia varten. Komponentin avulla kehittäjät voivat seurata tuotannonsuunnittelujärjestelmän toimintaa asiakkaan tuotantoympäristössä. Suunnitteluvaiheessa valittiin tietojen keräämiseen käytettävä metriikkakirjasto. Komponentti kerää suorituskykytietoja käyttäjän instanssista ja lähettää metriikkadataa valittuun Microsoftin Application Insights -palveluun. Palvelussa kerättyjä tietoja voidaan ryhmitellä ja visualisoida. Analysoidun datan avulla järjestelmätoimittaja voi tarkastella järjestelmän suoriutumista etänä ja tehdä tarvittavia toimenpiteitä, mikäli ongelmakohtia tai pullonkauloja ilmenee. Ohjelmakoodiin tehtävien muutosten minimoimiseksi monitorointiominaisuudet toteutettiin aspektiohjelmointia käyttäen.

ABSTRACT

EERO OINAS: Performance Instrumentation of Production Planning System
Tampere University of Technology
Master of Science Thesis, 45 pages
December 2015
Master's Degree Programme in Information Technology
Major:
Examiner: Professor Mikko Tiusanen
Keywords: production environment, instrumentation, Application Insights, aspect-oriented programming

Performance instrumentation of desktop applications in the actual operating environment provides developer information on the quality and usability of the actual operating conditions of the software. However, instrumentation outside the development environment requires a lot of developer resources, since adding the data collection operations often require significant changes to an existing code base.

SW-Development is a software company, focused on developing production planning applications for manufacturing industries. Their production planning application, SWD PES, provides tools for production planning and optimization. The application utilizes large amount of database data and carries out heavy calculations. The application is still under development. As a part of the continual improvement of the product it is important to be able to measure performance outside the development environment.

In this master's thesis an instrumentation component was designed and developed to provide developers a tool to monitor application performance in customer's production environment. Appropriate metrics library for collecting data was chosen during design phase. The component collects metrics data from the user application instance and sends data to Microsoft Application Insights service. The service can be used to group and visualize collected data. By analyzing the data, developers can examine the application performance remotely and make necessary procedures in case problems or performance bottlenecks are found. Aspect-oriented programming was used to minimize the changes in application code.

ALKUSANAT

Tämä diplomityö tehtiin tamperelaiselle ohjelmistoyritykselle SW-Development Oy:lle. Työssä käsitellään yrityksen tuotannonsuunnittelujärjestelmän suorituskykymonitoroinnin kehittämistä. Haluan kiittää kollegoitani Henri Wiklundia ja Mikko Kunnaria ohjauksesta ja opastuksesta niin diplomityöni kuin työurani aikana.

Haluan kiittää myös työni tarkastajaa professori Mikko Tiusasta hyvistä neuvoista ja työni ohjaamisesta oikeaan suuntaan. Kiitokset myös kaikille ystävilleni ja perheelleni, jotka ovat tukeneet minua kirjoitusprojektin aikana. Erityiskiitokset haluan antaa puolisolleni Marialle tuesta, kannustuksesta ja kärsivällisyydestä.

Tampereella 20. marraskuuta 2015

Eero Oinas

SISÄLLYSLUETTELO

TERMIT JA LYHENTEET

APS	ERP-järjestelmien täydennykseksi kehitetty tuotannon hienokuormitusohjelmisto.
Aspekti	Hajaantuneen ja sotkeutuneen ohjelmakoodin modularisoimiseksi tehty ohjelmayksikkö.
C#	Microsoftin kehittämä olio-ohjelmointikieli.
HTTPS	Hypertext Transfer Protocol Secure on HTTP-protokollan ja SSL/TLS-protokollan yhdistelmä, jota käytetään salatussa verkkoliikenteessä.
JSON	JavaScript Object Notation. Avoimen standardin tekstipohjainen tiedonvälitysformaatti verkkosovelluksiin.
Liitoskohta	Ohjelmakoodin sijainti, johon aspekteja voidaan liittää.
Läpileikkaus	Vaatimus tai näkökulma, joka on ratkaistu useammassa ohjelmayksikössä.
Metriikka	Matemaattinen kuvaus ohjelmakokonaisuuden osasta numeeriseksi arvoksi.
MsSQL	Microsoftin kehittämä relaatiotietokantojen hallintajärjestelmä.
.Net	Microsoftin kehittämä ohjelmistokomponenttikirjasto. Kirjasto sisältää palveluita ja komponentteja työpöytä-, mobiili- ja webkehitystä varten.
SWD PES	Planning Effieciency System. SW-Development Oy:n kehittämä tuotannonsuunnittelujärjestelmä.
UML	Unified Modeling Language on graafinen mallinnuskieli järjestelmäkehitykseen ja -suunnitteluun.

1. JOHDANTO

Nykyisin ohjelmistojärjestelmien kehitys tapahtuu usein ketteriä menetelmiä käyttäen. Ketterien menetelmien avulla ominaisuuksia pyritään toteuttamaan pienissä osissa, jolloin tilaavalle asiakkaalle pystytään toimittamaan jo kehitysvaiheessa osatoimituksia järjestelmästä. Ohjelmiston kehityksen aikana suorituskykymittauksia tehdään usein vain kehitysympäristössä ja paikallisia tietokantoja käyttäen. Tämä saattaa antaa kehittäjille liian positiivisen käsityksen järjestelmän suorituskyvystä tuotantoympäristössä. Kehitysympäristöjen erot tuotantoympäristöön aiheuttavat kehittäjille vaivaa luoda identtisiä ympäristöjä asiakkaan kanssa. Nopeammat verkkoyhteydet, tietokannan sijanti ja virtuaalityöpöytäratkaisujen vuoksi jaetut resurssit saattavat aiheuttaa suuriakin eroja ohjelman suoriutumiseen eri ympäristöissä. Suorituskykytestausta ja -monitorointia tulisi kuitenkin tehdä kehitysaikana kehitysympäristön lisäksi myös tuotantoympäristössä.

Tämä työ kehittää suorituskykymonitorointia tuotannonsuunnittelujärjestelmässä. Suorituskykymonitorointia on tarkoitus pystyä tekemään myös asiakasyrityksen tuotantoympäristössä normaalin testauksen yhteydessä. Työ tehdään tamperelaiselle ohjelmistoyritykselle, SW-Developmentille. Tavoitteena on luoda monitorointi-komponentti yrityksen jo olemassa olevaan tuotannonsuunnittelujärjestelmään Planning Effieciency Systemiin (SWD PES) (SW-Development 2015). Toteutettavan komponentin merkittävimpinä kriteereinä ovat modulaarisuus, tarkasti määritelty monitorointidata, tietoturva ja instrumentoinnin suorituskyky.

Monitorointi antaa mahdollisuuden kehittäjälle seurata järjestelmän toimintaa asiakkaan ympäristössä erillisen työkalun avulla. Monitorointi on toteutettava siten, että ennaltamääriteltyjä kohtia ohjelmassa voidaan seurata myös kehitysympäristön ulkopuolella ja siten, että monitoroinnin avulla kerätty data on mahdollista lähettää keskitetysti ohjelmistotoimittajan palvelimelle analysointia varten.

Monitoroinnin tarkoitus on parantaa asiakastyytyväisyyttä parantamalla ohjelman laatua löytämällä mahdollisia ongelmakohtia ja suorituksen pullonkauloja korjaustoimenpiteitä varten. Monitoroinnin on varmistettava tarvittava tietoturva ja salaus ulospäin lähetettävään dataan, koska monitorointi suoritetaan yritysten tuotantoympäristössä. Yrityssalaisuuksien ehdoton salassapito on asiakkaalle usein olennaisen tärkeää.

Kerättävän datan täytyy olla tarkasti määriteltyä, jotta sitä pystytään analysoimaan kokonaisuutena. Ongelmien paikallistamiseksi datan täytyy olla helposti tulkittavissa ja lajiteltavissa isostakin datamäärästä. Monitoroinnin tarkoitus on tutkia suori-

tuskykyongelmia ohjelman sisällä ja näin se ei itse saa tuottaa huomattavia suoritus-kustannuksia järjestelmää mitatessa. Toteutus pyritään pitämään hallittavana kokonaisuutena ottaen huomioon yrityksen laatuvaatimukset. Valmiiseen koodipohjaan tarvittavien ohjelmistomuutosten minimoimiseksi komponentti toteutetaan aspekti-ohjelmointia apuna käyttäen.

Aspektiohjelmointi tarjoaa komponenttiin tavan toteuttaa hajautuvaa monitorointikoodia yhdessä ohjelmayksikössä, ilman kopioitua koodia (Bergmans & Lopes 1999). Komponentti käyttää toteutuksessaan käännösaikaista aspektipunontaa, eli kaikki aspektiohjeet käännetään mukaan järjestelmän koodiin. Toinen vaihtoehto olisi käyttää ajoaikaista punontaa, jota esimerkiksi Aprobe-instrumentointisovellus (Cole 2004) käyttää. Ajoaikainen punonta aiheuttaa pieniä tietoturvariskejä, koska punonta perustuu siihen, että suoritettavaan järjestelmään linkataan toimintoja ulkoisesta ohjelmasta. Näin ei pystytä täysin varmistumaan siitä, mitä ulkoinen sovellus järjestelmän tiedoilla tekee.

Monitorointi toimii instrumentointiperiaatteella, eli mittaus suoritetaan heti mitattavan toiminnon tapahtumahetkellä. Näin metriikkatietoja saadaan kerättyä tarkalla tasolla operaatioittain. Metriikkatietojen keräämiseen käyteään Microsoftin kehittämää Application Insights -palvelua (Wills 2015), joka käyttää hyödykseen Microsoft Azure -alustaa (Microsoft Azure 2010).

Ensiksi työssä käydään läpi ohjelmiston suorituskyvyn termejä ja miten suorituskykyä voidaan parantaa, jonka jälkeen tarkastellaan aspektiohjelmointia ja -suunnittelua. Neljännessä luvussa tutkitaan työn lähtötilannetta, eli millaiseen järjestelmään suorituskykymonitorointia ollaan tekemässä ja mitä toteutettavassa komponentissa tulee ottaa huomioon. Suunnitteluvaiheessa vertaillaan myös tiedon keräämiseen käytettäviä metriikkakirjastoja, joista yksi valitaan työn toteutusta varten. Luvussa viisi esitellään toteutettu ratkaisu tuotannonsuunnittelujärjestelmän suorituskykymonitorointiin, ja se, miten suorituskykydataa kerätään ja lähetetään jatkokäsittelyä varten. Tuloksissa tarkastellaan, miten toteutus vastaa määriteltyjä vaatimuksia ja käydään läpi mahdollisia jatkokehitysehdotuksia.

2. SOVELLUKSEN SUORITUSKYKY

Ohjelmistojen yhteydessä suorituskyvyllä tarkoitetaan ohjelmiston käyttäytymisen ajallista täsmällisyyttä suhteessa asetettuihin tavoitteisiin (Smith & Williams 2001). Ohjelmiston suorituskykyyn voivat vaikuttaa kaikki ohjelmiston toteutukseen sidoksissa olevat asiat. Näitä ovat suunnittelu, ohjelmakoodi ja ohjelman suoritusympäristö. (Woodside et al. 2007)

Monitoroinnin ja sen avulla tehdyn analyysin avulla pyritään ylläpitämään ohjelmiston laatu halutulla tasolla. Sovelluksen suorituskyvyn taso on yleensä määritelty jo ohjelmistoprojektin alussa vaatimusmäärittelydokumentissa. Järjestelmän suorituskykyä on tärkeä ylläpitää vaaditulla tasolla. Kriittisissä järjestelmissä suorituskykyongelmat voivat aiheuttaa suuriakin vahinkoja laitteille ja henkilöille. Järjestelmän suorituskykyongelmat voivat myös kuormittaa järjestelmätoimittajan asiakassuhteita, jos vastaavia paremmin suoriutuvia vaihtoehtoisia järjestelmiä on tarjolla muualla. Huonosti suoriutuvat järjestelmät ovat selkeä haitta liiketoiminnalle.

2.1 Suorituskyvyn mittaaminen

Suorituskyvyn mittaukset rajoittuvat usein vain kehitysympäristöön ja järjestelmän testausvaiheeseen. Näiden asioiden mittaaminen on tärkeää myös tuotantokäytössä olevassa järjestelmässä. (Woodside et al. 2007) toteavat, että suorituskyvyn mittaaminen tarvitsee usein strategian, jonka avulla voidaan määritellä mitä asioita mitataan ja milloin mittauksen tulee tapahtua. Myös mittausolosuhteet on hyvä ottaa huomioon strategian määrittelyssä. Testiolosuhteissa suorituskykytesteissä voidaan käyttää suurta määrää dataa, jonka avulla pystytään simuloimaan laajojen järjestelmien datamääriä. Jossain tapauksissa tämä ei kuitenkaan ole mahdollista ja testiympäristö jää tuotantoympäristöön verrattuna vajavaiseksi.

Laaja testausympäristö voi luoda kuitenkin suuria kustannuksia kehittäjälle. Suurien datamäärien luominen siten, että testaus pystytään suorittamaan aina samanlaisessa ympäristössä vie aikaa ja näin luo ylimääräisiä kustannuksia. Tässä tapauksessa automaattinen datan luominen voisi mahdollistaa synteettisen datan tuottamisen, jolloin testaus pystytään suorittamaan identtisessä ympäristössä. (Ndem et al. 2011)

Tehtyjen mittausten tarkoituksena on lisätä järjestelmätason ymmärrystä ja verrata saatuja tuloksia samanlaisten ohjelmistojen vastaaviin mittauksiin. Mittausten avulla saadaan lisäksi määriteltyä parempia suorituskykymalleja, joiden avulla pystytään luomaan parempia arvioita ohjelmiston työkuormasta ja resurssitarpeesta. Kun kehitettävän ohjelmiston suorituskykyä mitataan jatkuvasti, pystytään ohjelmistosta

tunnistamaan suorituskykyyn liittyviä sudenkuoppia ja mahdollisia parannuskohteita koko ohjelmiston elinkaaren ajan. (Woodside et al. 2007)

Koko järjestelmän tutkiminen kokonaisuutena ei ole monitoroinnin kannalta mielekästä. Sen sijaan yksittäisten moduulien ja komponenttien erillinen monitorointi antaa paremman tuloksen, sillä kerättyjä tietoja ryhmittelemällä saadaan huomattavasti parempi näkemys oikeista ongelmien aiheuttajista. (Mazurov & Couturier 2012)

Suorituskyvyn mittaaminen voidaan jakaa kahteen kategoriaan: instrumentointiin ja profilointiin. Ohjelman profilointi on otospohjaista mittaamista. Se tarkastelee ohjelmaa tasaisin väliajoin ja kerää dataa ohjelmasta. Profilointimenetelmien tarkoituksena on tuottaa mahdollisimman vähän ylimääräistä kuormaa mitattavalle ohjelmalle. Suorittimen käyttö ja kulutettu muisti ovat esimerkikkejä asioista, joita käyttöjärjestelmät mittaavat suoritettavista ohjelmista määrätyin väliajoin. Koska profiloinnin näytteenottotaajuus voi olla pieni, mittauksen tarkkuus voi kärsiä. Näytteenottopisteiden välissä voi tapahtua resurssikuormitusta, joka ei näykään kuormapiikkinä. Pieni näytteenottotaajuus ei toisaalta aiheuta merkittäviä kustannuksia ohjelmiston suoritukseen.

Instrumentoinnin ideana on lisätä mitattavaan sovellukseen keräilijöitä (probes), joiden avulla pystytään mittaamaan ja tallettamaan erilaisia arvoja ohjelman suorituksesta määrättyjen asioiden tapahtumahetkellä (Woodside et al. 2007). (Smith & Williams 2001) mukaan instrumentointi saattaa aiheuttaa purskeista rasitusta mittauksen aikana, koska suuria määriä mitattavia kutsuja voi tapahtua pienen aikavälin sisällä, kuten jos mitattava metodi on silmukan sisällä. Vastineeksi instrumentoinnin tarkkuus on suurempi, sillä mittauspiste on aina mitattavan asian suoritushetki. Yleensä instrumentoitavia ovat metodikutsujen määrä, tietokantoihin tehtävät palvelupyynnöt ja tilanvaraukset.

Instrumentointi tarvitsee myös suunnitelman, jossa tunnistetaan mittauspisteet joihin keräilijät lisätään. Kun mittauspisteet on asetettu, tarvitsee ohjelmaa suorittaa siten, että näihin mittauspisteisiin päästään. Mitattavat toiminnot voidaan määrittää testaussuunnitelmassa ja tehdä automatisoidut testit, jotta mittauspisteeseen varmasti päästään. (Huang 1978)

Instrumentointi toteutetaan usein vain kehitysympäristössä. Ohjelmien ja kirjastojen julkaisukäännöksissä instrumentointi on otettu pois päältä. Tämä saattaa aiheuttaa ongelmia asiakasympäristössä tehtävään instrumentointiin, mikäli tiettyä ongelmaa tai virhettä yritetään toistaa ilman instrumentoinnin antamia lisätietoja. Tämän vuoksi ohjelmasta saatetaan tehdä kaksi eri käännöstä, jossa toisessa instrumentointi on kytketty päälle. (Cole 2004)

2.2 Suorituskyvyn parantaminen ja analysointi

Ohjelmiston suorituskyvyn hallintaa (Software Performance Engineering, SPE) käytetään usein ohjelmiston suorituskyvyn mittaamiseen tai sen parantamiseen. Sen tarkoituksena on keskittyä suurimpiin suorituskykyongelmiin sovelluksessa. SPE jaetaan viiteen tarkasteltavaan elementtiin: järjestelmäoperaatioihin, käyttäytymiseen, työkuormaan, järjestelmän rakenteeseen ja resursseihin. SPE voidaan katsoa myös osaksi ohjelmistotestausta. (Woodside et al. 2007)

Mikäli joidenkin käyttötapauksen vasteajat ovat liian suuria tai käytettävä resurssi ruuhkautuu, järjestelmää joudutaan analysoimaan ja tutkimaan, mitkä toiminnot aiheuttavat pullonkauloja järjestelmässä. Suorituskykyä voidaan parantaa nostamalla yleisesti laitteiston tehoja, jolloin laitteisto pystyy suorittamaan pyyntöjä nopeammin tai useampia pyyntöjä samanaikaisesti. Mikäli pullonkaulojen havaitaan johtuvan jonkin tietyn pyynnön suorituksesta, kannattaa tämän toiminnon suorituspolkua analysoida. Ongelmakohtien löydyttyä ohjelmakoodia voidaan järjestellä uudelleen ja optimoida tehokkaammaksi esimerkiksi soveltuvampia tietorakenteita käyttäen. (Koliai et al. 2010)

Suorituskykyanalyysissä halutaan löytää mahdolliset ongelmat jo järjestelmän kehitysvaiheessa, mikä vähentää jatkokehityksen ja korjauksien kustannuksia. Näin pystytään vähentämään koko projektin läpivientiaikaa ja kokonaiskustannuksia. Mittareiden avulla voidaan analysoida ohjelman suorituskyvyn ylärajaa, kapasiteettia. Mittauskohteet tulee määrittää suorituskykystrategiassa ja käytetyissä ohjelmistomalleissa (software patterns) siten, että järjestelmästä voidaan helposti määrittää mitattavat ja analysoitavat kohteet. (Koliai et al. 2010)

Analysoinnin avulla mahdollisten pullonkaulojen tai ongelmakohtien löydyttyä voidaan tehdä päätöksiä, miten ongelma halutaan korjata. Pullonkaulan löydyttyä ongelmaa voidaan validoida vielä tarkemmin vähentämällä ruuhkautuvaa resurssia, jonka jälkeen mittaukset tehdään uudelleen. Näin pystytään havaitsemaan kriittisen resurssin vaikutus kokonaiskuormaan ja myös määrittämään resurssin lisätarve, joka tarvitaan järjestelmän ruuhkautumisen estämiseksi. Tämän jälkeen kriittistä resurssia voidaan lisätä vain tarpeellinen määrä, tai ainakin siten, että rahalliset lisäkustannukset resurssin kasvattamiselle voidaan minimoida. (Cole 2004)

3. ASPEKTIOHJELMOINTI

Olio-ohjelmointi pyrkii modularisoimaan toiminnot ja toiminnallisuuden niitä vastaaviin ohjelmayksiköihin. Nämä saattavat kuitenkin jakaa yhteistä toiminnallisuutta muiden yksiköiden kanssa, minkä vuoksi olio-ohjelmointimallin mukaan toteutetut toiminnot voivat johtaa hajaantuneeseen (scattered) ja sotkeutuneeseen (tangled) koodiin useissa yksiköissä. Näitä jaettuja yhteisiä toiminnallisuuksia kutsutaan läpileikkaaviksi vaatimuksiksi.

Vaikka olio-ohjelmointikielet tarjoavat hyviä keinoja modularisointiin ja yksityiskohtaiseen hallinnointiin, ne eivät kuitenkaan ratkaise kaikkia modularisoinnin ongelmia. Luokat ja kirjastot eivät tarjoa mahdollisuutta sotkeutuneen koodin minimointiin, eivätkä myöskään anna mahdollisuutta lisätä, poistaa ja muokata ohjelman toimintaa globaalisti. Näiden ominaisuuksien toteutus ja hallinta tyypillisesti hajautuu useaan ohjelmayksikköön, mikä rikkoo oliosuunnittelun periaatteita. Läpileikkaavien ominaisuuksien hajaantuminen pyritään hallitsemaan aspektiohjelmoinnin avulla. Läpileikkaavia vaatimuksia ovat virhekäsittely, transaktiot, tietoturva ja samanaikaisuuden hallinta. (Murphy & Schwanninger 2006; Bergmans & Lopes 1999)

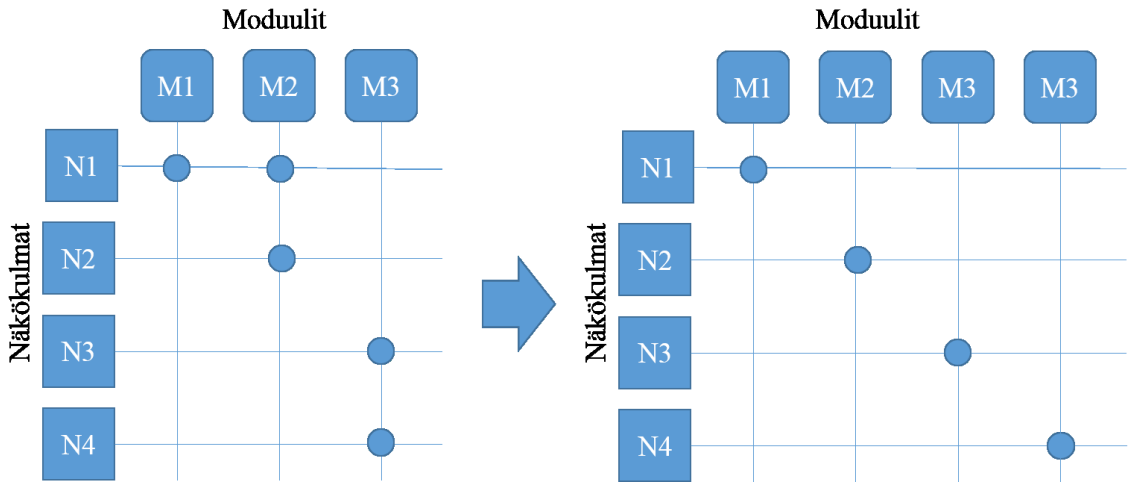
Oliosuunnittelun ideana on pilkkoa vaatimukset pieniin yksiköihin, luokkiin, jotka vastaavat omasta logiikastaan ja tarjoavat rajapinnat ulospäin muille ohjelmayksiköille. Aspektiperustaisen suunnittelun avulla oliosuunnittelun luomaa yksikköjaon abstraktiotasoa voidaan nostaa ja useille luokille yhtenäiset ominaisuudet voidaankin toteuttaa aspektiperustaisen suunnittelun avulla ilman kopioitua koodia. Vaikka kehittäjä suunnittelisi oliototeutuksen huolellisesti, jotkut vaatimukset päätyvät silti hyvin monikäyttöiseksi tai epämodulaariseksi. Tämä aiheuttaa suuria määriä riippuvuuksia ja sotkeutumista luokkien välillä. (Murphy & Schwanninger 2006; Abdelzad & Aliee 2011)

Aspektiohjelmointi on ohjelmointiparadigma, jonka ei ole tarkoitus korvata olio-ohjelmoinnin ohjelmointiparadigmaa, vaan luoda uusi tapa liittää moduuleita ohjelmaan. Sen tärkeimpänä kohteena ovat paikat, joissa hajaantunut ohjelmakoodi ei anna lisäarvoa liiketoimintalogiikalle. Ideana on vähentää yhtenevää koodipohjaa siirtämällä geneerinen toiminnallisuus omaan yksikköön, aspektiin (aspect), joka voidaan suorittaa kaikissa sitä tarvitsevissa toiminnoissa ilman kopioitua koodia. Tämä johtaa pienempään ja helpommin ylläpidettävään koodipohjaan, joka puolestaan vähentää virheitä koodissa ja parantaa koodin laatua ja selkeyttä. (Murphy & Schwanninger 2006)

Yksittäisen aspektin tarkoituksena on toimia ratkaisuna läpileikkaavien vaatimusten aikaansaamiin ongelmiin (Abdelzad & Aliee 2011). Aspekti koostuu ohjeista (advice) ja liitoskohdista (join point), joiden avulla ongelma pyritään ratkaisemaan. Tarkoitus on

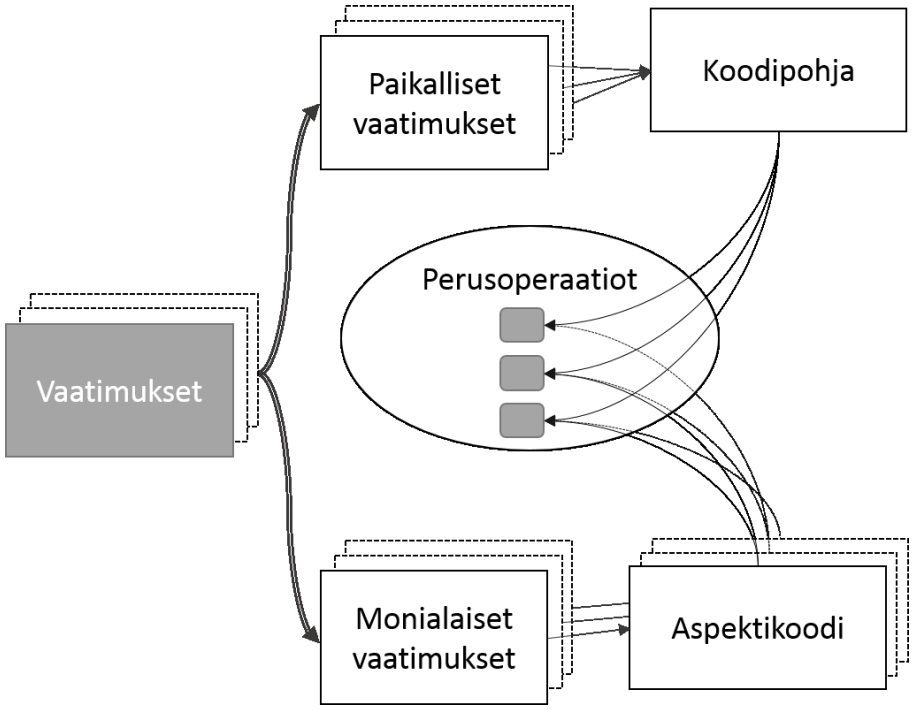
toteuttaa läpileikkaava vaatimus siten, että sitä voidaan käyttää kaikkialla kyseistä toimintoa tarvitsevissa ohjelmayksiköissä. (Kiczales et al. 1997)

Olio-ohjelmoinnissa perusoperaatioiden kommunikointitapa olioiden kanssa määrittää koodin rakenteen ja jaon luokkiin. Aspektiohjelmoinnissa ja -suunnittelussa tämä on kuitenkin hieman monimuotoisempaa. Suunnittelija määrittelee toimialueen ja siihen liittyvät vaatimukset, jotka ohjelmiston tulee täyttää. Suunnittelun ja vaatimusten jakamisen apuna voidaan käyttää näkökulmien erottelu -periaatetta (separation of concerns; Aksit et al. 1996). Se perustuu ongelman ratkaisuun jakamalla ongelma pienempiin osaongelmiin. Kun osaongelmien ratkaisut yhdistetään, saadaan alkuperäisen ongelma ratkaisu. Kuvassa 1 on esitetty näkökulmien erottelu -periaatteen käyttö. Mikäli moduuli ratkaisee usemman näkökulman ongelman, se aiheuttaa sotkeutumista. Hajaantuminen puolestaan aiheutuu siitä, jos näkökulmaan on ratkaisu useammassa moduulissa. Ideaalitilanteessa pyritään siihen, että yksi moduuli ratkaisee yhden näkökulman.



Kuva 1. *Näkökulmien erottelu -periaate.*

Suunnitteluvaiheessa voidaan määrittää, ratkaiseeko näkökulma paikallisia vaatimuksia vai onko mukana myös läpileikkaavia vaatimuksia. Tämän perusteella voidaan suunnitella koodipohjan ja aspektikoodin jako siten, että paikalliset vaatimukset hoidetaan koodipohjassa ja geneeriset läpileikkaavat vaatimukset ratkaistaan aspektikoodin puolella. Kuvassa 2 esitetään määriteltyjen vaatimusten jako paikallisiin ja läpileikkaaviin vaatimuksiin. Jaon perusteella toteutettavat koodit jaetaan koodipohjaan ja aspektikoodeihin. Aspektikoodit vastaavat läpileikkaavien vaatimusten toteutuksesta siten, että useampi aspekti voidaan yhdistää yhteen perusoperaatioon. Aspektin tehtävä on vastata yhteen tai useampaan läpileikkaavaan vaatimukseen. Paikalliset vaatimukset hoidetaan puolestaan perusohjelmassa. (Bergmans & Lopes 1999; Kiczales & Mezini 2005)



Kuva 2. *Aspektiperustaisen suunnittelun periaate.*

Aspektiperustaisessa suunnittelussa luokkien ei tarvitse olla sidoksissa rajapintojen kautta tiettyihin aspekteihin tai toisin päin. Tämä helpottaa kehittäjien työtä, koska he joutuvat sitoutumaan ainoastaan luokkien rajapintoihin, eikä luokkien tarvitse rikkoa modulaarisuuttaan tarjotakseen rajapinnan kautta monialaisten ominaisuuksien edellyttämiä toimintoja. Nykyaikaiset olio-ohjelmointikielet, kuten C++, C# ja Java, mahdollistavat aspektien käytön niille luotujen kirjastojen avulla. (Murphy & Schwanninger 2006)

3.1 Liitoskohta ja ohje

Liitoskohta määrittää sijainnin ohjelmakoodissa, johon voidaan kiinnittyä ja missä ylimääräistä koodia eli ohjeita suoritetaan. Aspekti voi määrittää useita liitoskohtia toteutuksessaan. Näitä ovat esimerkiksi metodin suoritukseen tulo, suorituksen jälkeinen hetki tai suorituksen aikana tapahtuva poikkeus. Myös muuttujatason muutoksiin voidaan lisätä toiminnallisuutta, mikäli käytettävä teknologia sallii näiden muutosten seuraamisen.

Aspektin sisältämät ohjeet ovat kokoelma toimintoja ja ylimääräistä ohjelmakoodia, joita aspektin tulee suorittaa määritellyssä paikassa. Liitoskohtamääritys kertoo, milloin ohje suoritetaan. Ohjeet toteutetaan aspektin sisälle, joten ohje voi sisältää omia paikallisia muuttujia tai käyttää aspektin näkyvyysalueen muuttujia omiin toimintoihinsa. Esimerkiksi funktiotason suoritusajan mittaamisessa funktion jälkeen suoritettava ohje voi käyttää hyväkseen funktion alussa muistiin otettua kellonaikaa. (Jacobson & Ng 2005)

Liitoskohtamäärityksen (pointcut) avulla on päätetty, mihin liitoskohtaan toiminta sidotaan. Aspektissa voidaan myös itse määritellä suoritetaanko koodi ennen vai jälkeen liitoskohdan. Näiden määritysten avulla aspektin ohjeet voidaan sitoa yhteen tai useampaan liitoskohtaan ohjelmassa. (Kellens et al. 2006)

3.2 Aspektin punonta

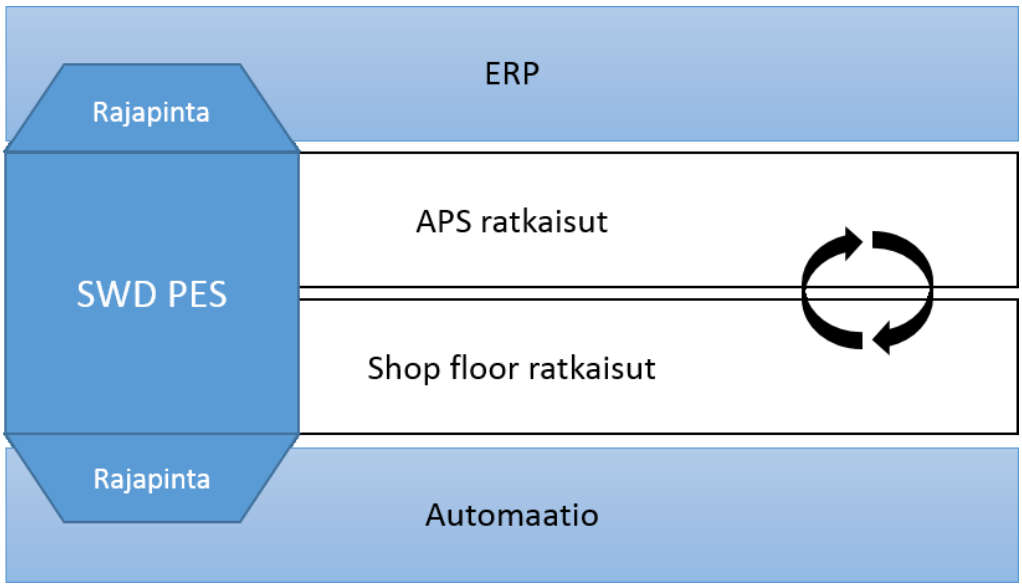
Aspektien punonta (aspect weaving) luo lopullisen toteutuksen ja esitysmuodon ohjelmasta tai sen komponentista, johon aspekteja on lisätty. Aspektin punoja toteuttaa tämän esitysmuodon suorittamalla aspektien määrittämän koodin suoritusaikana liitoskohdissa tai kääntämällä aspektikoodit valmiiksi suoritettavaan ohjelmaan tai kirjastoon.

Suoritusaikana tehtävä punonta voidaan toteuttaa käyttämällä kielen tarjoamia reflektio-ominaisuuksia. Liitoskohtamääritteinen malli (point cut model) pitää kirjaa ohjelman funktiokutsuista, niitä kutsuvista luokista ja funktion parametreista. Näiden tietojen avulla aspektipunonnassa sidotaan aspektiohje oikeisiin liitoskohtiin käyttäen alkuperäisiä parametreja. Suoritusaikana tehty punonta voidaan toteuttaa metaohjelmana, jota kutsutaan pääohjelman metodien kutsumishetkellä. Aspektiohjelma käyttää kutsun metatietoja ja parametreja toteuttaakseen aspektikoodit määritellyllä tavalla. (Kiczales et al. 1997)

4. SWD PES MONITOROINTIKOMPONENTTI

Enterprise Resource Planning (ERP) -toiminnanohjausjärjestelmät toimivat liian karkealla tasolla. Niiden avulla ei pystytä reagoimaan riittävän nopeasti ja tarkasti tuotannon jokapäiväisessä toiminnassa tapahtuviin tilanteen muutoksiin (Thompson 2006). Esimerkiksi keskeisen tuotantokoneen hajoaminen saattaa aiheuttaa tarpeen tuotantosuunnitelman uudelleen järjestämiseen. Tästä syystä on kehitetty Advanced Planning and Scheduling (APS) -ratkaisuja, jotka soveltuvat tuotantosuunnitelman nopeaan iterointiin ja voidaan räätälöidä vastaamaan asiakaskohtaisia eritystarpeita.

SW-Developmentin Planning Efficiency System (SWD PES) on valmistavan teollisuuden tarpeisiin kehitetty tuotannonsuunnittelujärjestelmä. Järjestelmä tarjoaa APS -ratkaisuja teollisuuteen, kuten tuotannonsuunnittelu- ja tuotannonoptimointityökaluja, toimitusketjun ja asiakkaiden hallintaa sekä lattiatason Shop Floor -ratkaisuja. SWD PES pyrkii toimimaan saumattomasti asiakkaiden valmiiden ERP -järjestelmien kanssa. Kuvassa 3 on kuvattu SWD PES järjestelmän sijainti muihin teollisuuden järjestelmiin nähden. Tuotantosuunnitelmia pystytään optimoimaan järjestelmän optimointityökaluja käyttäen. Suunnittelijan on myös mahdollista tehdä kokeellisia skenaarioita turvallisessa hiekkalaatikkoympäristössä. (SW-Development 2015)

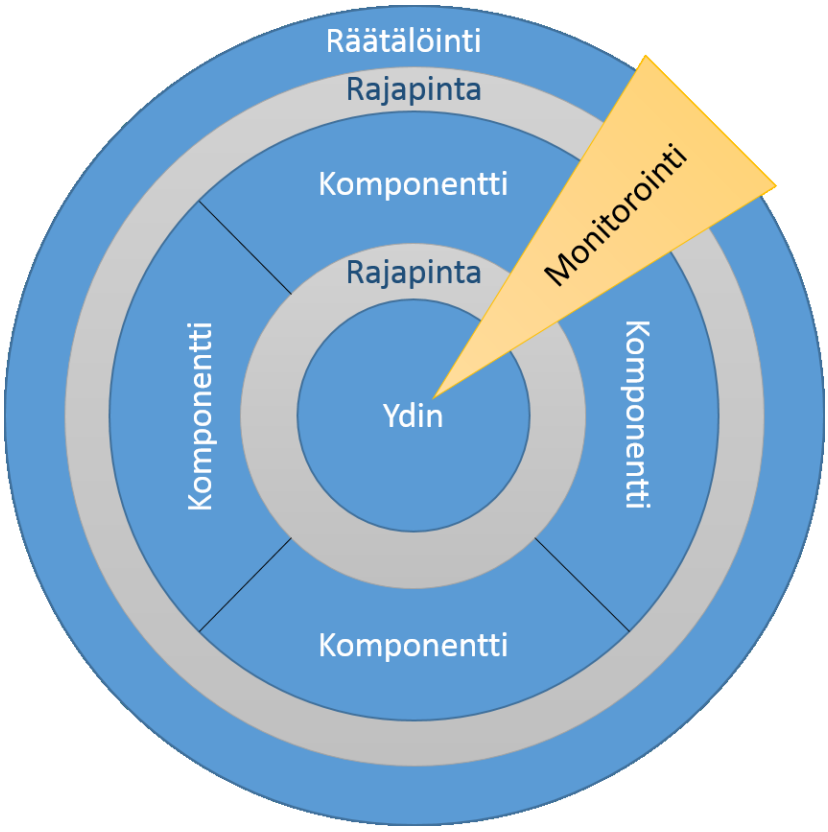


Kuva 3. *SWD PES järjestelmän suhteet muihin järjestelemiin*

Shop Floor -ratkaisut avaavat tuotantosuunnitelman työntekijöille. Ratkaisu tarjoaa työntekijöille tehtävälistoituksen, jossa tehtäviä pystytään seuraamaan erilaisin kuittauksin aloituksesta tehtävän lopetukseen saakka. Shop Floor -ratkaisuissa on myös

raportointityökaluja ja tehokkuusmittareita, joiden avulla työnlaatua pystytään seuraamaan. Nämä ratkaisut on myös mahdollista yhdistää asiakkaan valmiisiin automaatiojärjestelmiin, jotta SWD PES pystyy toimimaan kokonaisvaltaisena välikappaleena ERP-tuotannonohjaus- ja automaatiojärjestelmien välillä. (Peltola 2015)

SWD PES –alustan toteutus on komponenttipohjainen kirjastokokoelma, ja asiakkaille räätälöidyt ratkaisut ovat kokoelmia halutuista toiminnallisuuksista ja niitä toteuttavista komponenteista. Komponentit perustuvat kirjastoon Swd.Component (Peltola 2015). Tämä kirjasto tarjoaa ydinominaisuudet ja käyttöliittymäelementit komponenteille. Tässä työssä toteutettavan monitorointikomponentin ei kuitenkaan ole syytä edes viitata alustan peruskomponentteihin tai periä niitä, jotta toteutettava komponentti saadaan pidettyä eriytettynä alustasta. Näin monitorointikomponentti voidaan ottaa käyttöön myös muissa ratkaisuissa tarpeen vaatiessa. Kuvassa 4 on esitetty SWD PES ratkaisujen komponenttirakenne ja niiden väliset suhteet.



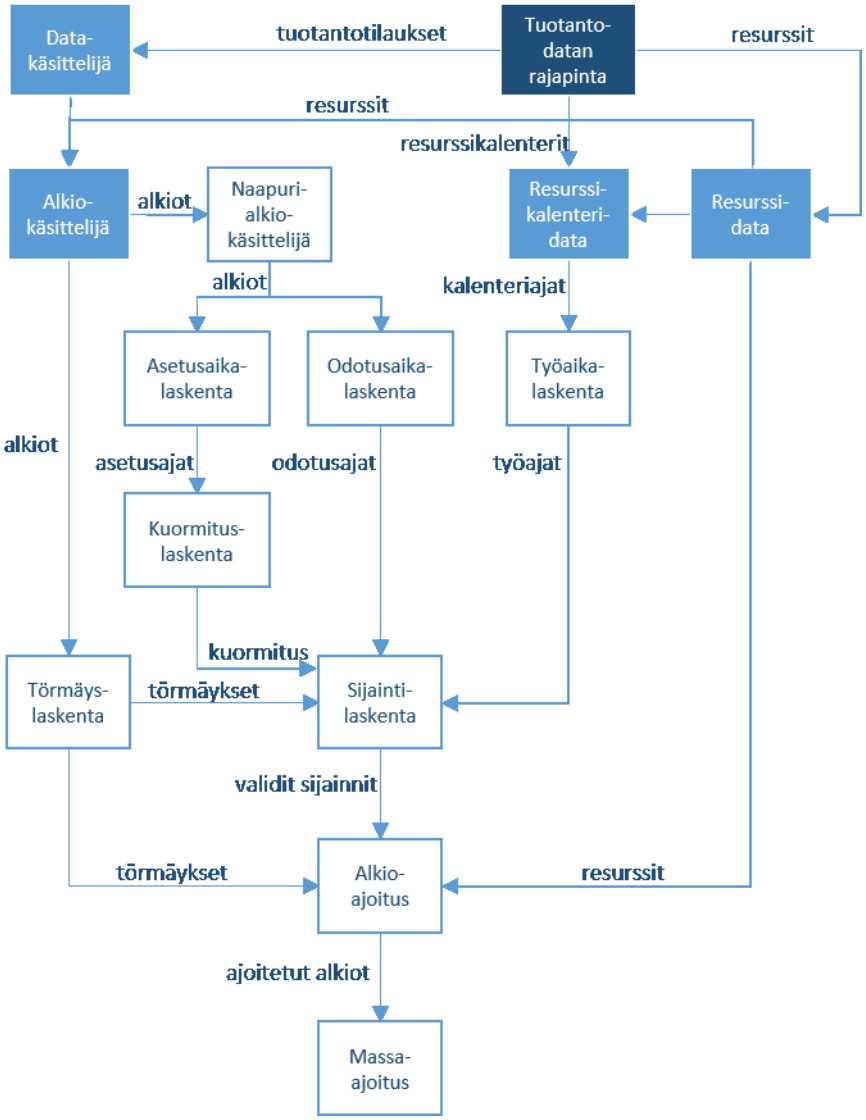
Kuva 4. *SWD PES-ratkaisujen komponenttirakenne ja suhteet.*

Komponentit tarjoavat räätälöintiin tarvittavat rajapinnat. Näiden rajapintojen avulla voidaan asiakasratkaisuihin valitut komponentit konfiguroida asiakkaalle sopiviksi ja ylläpitää perustoiminnallisuudet komponentin perustoteutuksessa.

Monitorointikomponentti tulee alustavasti vain SWD PES-järjestelmän käyttöön ja siksi komponentti toteutetaan vain kyseistä järjestelmää silmällä pitäen. Monitorointikomponentti integroidaan Swd.Component-kirjastoon, jotta alustan tarjoamia perustoimintoja pystytään instrumentoimaan. Monitoroitavia toimintoja ovat muun muassa komponenttien alustukset, käynnistykset ja näkymän päivitykset. Nämä toiminnot käyttävät usein suurta määrää tietokantadataa ja vaativat datan prosessointia näkymää varten. Tämän vuoksi kyseiset instrumentoitavat toiminnot ovat suorituskyvyn monitoroinnin kannalta hyvin oleellisia.

4.1 Suorituskyvyn mittaamisen tarve

APS-ratkaisujen keskeisiä toimintoja on suunnitelman aikataulutus. Myös SWD PES tarjoaa useita ajoitusalgoritmeja tuotantosuunnitelman automaattiseen ajoitukseen ja optimointiin. Näitä algoritmeja tarvitsee usein lisätä ja muokata asiakkaan tarpeiden mukaisiksi. Tästä aiheutuu haasteita ajoituksen suorituskyvyn mittaamiseen. Yhtenä esimerkkinä SWD PES -järjestelmän tarjoamista ajoitusalgoritmeista on tuotantosuunnitelman massa-ajoitus. Kuvassa 5 on kuvattu SWD PES -järjestelmän massa-ajoituksen rakenne ja laskentaan käytettäviä ohjelmayksiköitä. Massa-ajoitus jakautuu useaan laskentayksikköön, joiden avulla tuotantotilaukset kohdistetaan suunnitelmaan optimaalisesti. Ajoitus ottaa huomioon järjestelmässä määritetyt resurssikapasiteetit ja -kalenterit. Algoritmin avulla tuotannonsuunnittelija pystyy automatisoidun laskennan avulla parantamaan suunnitelmaa.



Kuva 5. *Massa-ajoitusalgoritmin periaate tuotantosuunnitelman optimointiin.*

Massa-ajoitus perustuu tuotantotilauksista tehtävien aika-alkioiden optimaalisen sijainnin määrittämiseen. Aika-alkiot kuvaavat yksittäistä tehtävää tuotannon aikana. Aika-alkiot haetaan käsittelijälle tuotantodatan rajapinnan avulla, joka toimii koostavana rajapintana tuotannonsuunnittelu komponenttien tarvitsemille datakyselyille. Tehtäville on määritelty kuormitettava resurssi ja mahdollisia muita ajoitusmääreitä. Resurssit kuormitetaan ottaen huomioon resurssikalenterit, joita resurssille on määritetty. Resurssikalenterit määrittävät avoimet työvuorot resurssille.

Eri laskentayksiköt muuttavat alkion sijaintia suunnitelmassa. Naapurialkiot tarkastetaan käsittelijän avulla ja tuloksena tulevat alkiot siirretään laskentayksiköille. Asetusaika- ja odotusaikalaskenta ottaavat huomioon töille määriteltyjä lisäaikoja töiden kuormituksen lisäksi. Alkion sijanti optimoidussa suunnitelmassa hallinnoidaan sijaintilaskennan aikana, jonka tulos siirretään ajoitustoiminnolle. Massa-ajoitus laskee kaikille alkioille optimaalisen sijainnin valitulle aikavälille käyttäen yksittäisen alkion ajoitustoimintoja.

Tällaisen laajan laskentakokonaisuuden suorituskyvyn seuranta mahdollistaa algoritmien suorituskyvyn parantelun. Toimiessaan tehokkaasti optimointityökalut tarjoavat tavan ylläpitää optimaalista tuotantosuunnitelmaa. Kun suorituskykyä seurataan metoditasolla, laajoistakin suoritusketjuista, kuten esitellystä massa-ajoituksesta, voidaan paikantaa suurimmat suorituskykyongelmat. Näin kehittäjiä voidaan ohjata korjaamaan ongelman juurisyyt.

4.2 Käytetyt tekniikat

SWD PES on toteutettu C# ja Microsoftin .Net teknologioita käyttäen. Monitorointi-komponentti toteutetaan myös samoja teknologioita käyttäen. C# (englanniksi lausuttuna C sharp) on Microsoftin kehittämä ohjelmointikieli. Se on vahvasti tyypitetty olio-ohjelmointikieli, joka kehitettiin .Net-konseptia varten 2000-luvun alussa. Kieli kehitettiin yhdistämään C++- (Lippman 2002) ja Delphi-kielien (Hodges 2014) tehokkuus ja Javan (Sabharwal 1998) helppokäyttöisyys. Se on tarkoitettu moderniksi ja yleiskäyttöiseksi olio-ohjelmointikieleksi, joka tarjoaa useita ohjelmistotekniikan periaatteita helpottavia ominaisuuksia. Kieli tarjoaa myös automaattisen roskienkeruun Java-kielen tapaan. (Ferguson et al. 2002)

.Net on Microsoftin kehittämä ohjelmistokehys. Kehystä voidaan käyttää yli 20 ohjelmointikielellä käyttäen Framework Class Library (FCL) -kerrosta. FCL tarjoaa muun muassa käyttöliittymien, tietokantayhteyksien ja verkkoyhteksien luomiseen tarvittavia työkaluja ja palveluita. .Net-kehys tarjoaa Windows, Windows Phone ja web-kehitykseen kirjastoja, jotka suoritetaan Common Language Runtime (CLR) -kerroksen päällä. CLR toimii virtuaalikoneena ja se tarjoaa palveluita tietoturvan, muistinkäytön ja poikkeuskäsittelyn tueksi. FCL ja CLR yhdessä muodostavat .Net-ohjelmistokehyksen. (Thai & Lam 2001)

4.3 Ohjelmistometriikat

Metriikka on kvantitatiivinen arvo, jonka avulla jotain ohjelmiston näkökulmaa mitataan. Se on matemaattinen kuvaus, jonka avulla pyritään muuttamaan ohjelmisto-kokonaisuuksia (entities) numeerisiksi arvoiksi. (Lincke et al. 2008)

Ohjelmistometriikoita käytetään usein ohjelmiston laadulliseen mittaamiseen. Niiden avulla tarkastellaan käytettyjen suunnittelumallien soveltuvuutta käyttötapaukseen. Tarkemmalla tasolla metriikoita voidaan käyttää esimerkiksi olio-ohjelmoinnissa ominaisten ohjelmayksiköiden tarkasteluun. Metodin koon, syklomaattisen kompleksisuuden (Karamath & Desharnais 2010) tai kommenttimäärän laadullinen tarkastelu on olennainen osa olio-ohjelmointiin liitettävää metriikkamittausta. (Rosenberg & Hyatt 1997)

4.3.1 Metrics .Net

Metrics .Net (Margarintescu 2015) on Javalle tehdyn metriikkakirjaston uudelleenkäännös Microsoft Windows ja .Net-ympäristöön. Kirjastoon on tehty myös useita lisäyksiä alkuperäiseen metriikkakirjastoon verrattuna. Kirjasto tarjoaa useita tapoja kerätä sovellustason tietoa .Net-ympäristössä suoritettavista ohjelmista. Metrics .Net soveltuu niin web- kuin työpöytäsovellusten instrumentointiin. Kirjastoa kehitetään aktiivisesti.

Kirjaston työkalut tarjoavat useita valmiita tapoja kerätä mitattavaa tietoa tuotantoympäristössä suoritettavista sovelluksista. Kirjasto kokoaa kerätyt tiedot tiedostoon ja pystyy myös lähettämään niitä HTTP-protokollan ylitse palvelimille. Toteutettavassa komponentissa kirjastoa käytetään tietojen keräämiseen, kokoamiseen loogisiksi kokonaisuuksiksi ja lähettämiseen eteenpäin jatkoanalysointia varten.

Kirjasto tarjoaa useita metriikoita erityyppisten tietojen keräämiseen:

- Kirjaston tarjoamilla ajastintoiminnoilla pystytään mittaamaan toimintojen kestoja. Ajastin on helppokäyttöinen ja soveltuu toteutettavan monitoroinnin tarpeisiin hyvin. Ajastin toimii omalla näkyvyysalueella alustamisen jälkeen. C#-koodissa ajastin on hyvä alustaa ja käynnistää using-lauseen sisällä, jotta ajastin lopettaa mittaamisen automaattisesti näkyvyysalueensa lopussa. Ajastimelle pystytään myös alustuksen yhteydessä antamaan toiminto, joka suoritetaan ajastinolion tuhoamisen yhteydessä.
- Laskuri on mittareista yksinkertaisin. Laskurin avulla pystytään laskemaan esimerkiksi suoritettavien toimintojen määrä. Laskurille voidaan alustuksen jälkeen kutsua lisäysmetodia, jonka avulla laskuri pitää kirjaa samoin luokitelluista lisäyksistä.
- Histogrammin avulla voidaan kerätä ajan suhteen arvoja. Arvot voidaan kerätä tietyltä aikaväliltä tai jatkuvasti. Kerätyistä arvoista lasketaan myös keskiarvo, minimi, maksimi ja mediaani.
- Mittari (gauge) on tarkoitettu tietyn mitattavan asian arvon, sen maksimien tai minimien seuraamiseen.
- Metriikkakirjasto tarjoaa kantaluokat sovelluksen tilatarkastuksille. Tilatarkastuksen avulla voidaan validoida eri resurssien ja ohjelmayksiköiden toimivuus. Sovelluskehittäjä voi itse periyttää tilatarkastuksia ja sisällyttää niihin omaa logiikkaa.

Metrics .Net kerää muistiin erän tietoja, jotka lähetetään yhtenä pakettina määritellyn ajan välein. Tämä kirjasto vaatii erillisen vastaanottavan järjestelmän, johon tiedot lähetetään. (Margarintescu 2015)

4.3.2 Application Insights

Application Insights (Wills 2015) on Microsoftin 2013 julkaisema analytiikkapalvelu, jonka avulla pystytään keräämään tietoa ohjelmistojen toiminnasta. Application Insights tarjoaa Application Performance Management (APM; Khanna et al. 2006) ja metriikkarajapinnat tiedon keräämiseen ja lähettämiseen suoraan Microsoftin Azure-palveluun (Microsoft Azure 2010).

(Wills 2015) mukaan Application Insights-palvelun avulla saadaan kerättyä tietoa ohjelman ongelmista, virheilmoituksista ja jäljittää käyttäjien tekemiä toimintoja. Palvelu onkin tarkoitettu kehittäjien aputyökaluksi, jonka avulla pystytään parantamaan ohjelman laatua ja käytettävyyttä. Palvelu on tarkoitettu kaikille web-, työpöytä- ja mobiilialustoille ja mahdollistaa samaa rajapintaa käyttäen tiedon keräämisen alustasta riippumatta.

Application Insights-palvelun rajapinta jakautuu metriikoiden, tapahtumien, kutsujen ja virheiden tietojen keräämiseen.

- Metriikan avulla pystytään keräämään yksittäisiä arvoja. Metriikkatiedot lähetetään nimiavaimen avulla, jonka avulla tietoja pystytään ryhmittelemään.
- Tapahtumat ovat manuaalisesti lisättyjä kohtia ohjelmassa. Yksittäisten tapahtumien määrää yhden käyttöistunnon aikana voidaan seurata.
- Kutsujen seurannan avulla pystytään keräämään tietoja tehdyistä palvelukutsuista ja lähettämään tiedot kutsun kestosta ja onnistumisesta.
- Virherajapinta tarjoaa poikkeustietojen ja virhetilanteiden tietojen keräämisen metriikkadataan.

Kaikkiin Application Insights-palvelun avulla kerättyihin tietoihin voidaan nimetä lisämääreitä, jotka liitetään aina kaikkiin metriikkakontekstiin kerättäviin tietoihin. Näihin tietoihin lisätään muun muassa kerättävän tietokoneen käyttöjärjestelmän versionumero, käyttäjätunniste ja sessiotunniste. Lähetettäviin tietoihin pystytään myös erikseen määrittelemään lisätietokenttiä, joita halutaan lähettää yksittäisen metriikkadatan mukana. Esimerkiksi yksinkertaisen metriikkatiedon yhteydessä resurssin statustietoja voidaan liitää tallennettavan metriikka-arvon lisätietoihin.

Tässä työssä nopea tapa kerätä suorituskykydataa on olennaista ja siksi Application Insights on tapaukseen soveltuva palvelu. Application Insights kerää dataa sisäisesti muistiin ja kykenee itsenäisesti lähettämään datan Azure-palvelimille, kun sitä on kerätty tarpeeksi. SWD PES-alusta on Microsoftin .Net teknologioilla toteutettu, siksi Application Insights-palvelun mukaan ottaminen suorituskykymonitorointiin on helppoa.

4.4 Vaatimukset

Toteutettavan komponentin on pysyttävä laadukkaana ja koko elinkaareltaan hallittavana kokonaisuutena. Ohessa on määritelty muutamia vaatimuksia, jotka pyritään toteutuksessa ottamaan huomioon.

Modulaarisuus

Komponentin on oltava modulaarinen siten, että metriikkakirjasto ja mahdollisesti myös käytetty aspektikehys ovat vaihdettavissa toiseen ilman suuria muutoksia. Komponenttiin luodaan abstrakti luokka metriikkakontekstia varten ja tämä voidaan periyttää jokaista käytettävää metriikkakirjastoa varten. Abstraktin luokan tarkoitus on luoda pohjatoteutus metriikkakontekstille ja pakottaa kehittäjä toteuttamaan tarvittavat metriikkafunktiot itse.

Modulaarisuus mahdollistaa jatkossa komponentin käytön myös muualla, missä suorituskykymonitorointia tarvitaan. Aspektikehyksen avulla luodut monitorointikeräilijät on mahdollista toteuttaa itse tai periyttää keräilijöitä komponenttiin luotujen peruskeräilijöiden tueksi.

Tarkasti määritelty data

Komponentin lähettämän datan pitää olla hyvin määritelty, jotta sen jatkokäsittely ja analysointi on helppoa. Tarkasti ja hyvin määritellyn datan avulla pystytään luokittelemaan kerättyjä tietoja ja tunnistamaan niistä järjestelmän ongelmakohdat. Luokittelu auttaa myös trendien seuraamisessa järjestelmää analysoitaessa.

Metriikkatiedot talletetaan avain-arvo -pareina. Avaimien tulisi olla yksilöiviä siten, että yksittäisiin tapauksiin liittyvät metriikka-arvot ovat tunnistettavissa helposti. Kun halutaan mitata kokonaista käyttötapauksen aiheuttamaa suorituspolkua, metriikkadatassa tulee olla määriteltynä tunnistettavasti käyttötapaus ja sen polku, johon kyseinen metriikka-arvo liittyy.

Komponentin lähettämän datan täytyy olla riittävää, jotta siitä saadaan tarpeeksi kattavaa analyysiä varten. Järjestelmän kohdat, joihin keräilijöitä laitetaan, on määriteltävä siten, että dataa kertyy tarpeeksi tutkittavista kohdista. Näiden avulla ongelmien löytäminen on helpompaa ja data pysyy kokonaisuudessaan käyttökelpoisena. Laajan datamäärän avulla trendit ovat tunnistettavempia ja ongelmien syy-seuraussuhteet ovat helpommin tunnistettavissa laajemmassa mittakaavassa.

Tietoturva

Komponentti lähettää dataa asiakkaan tuotantoympäristöstä verkon ylitse, joten datan lähettäminen pitää olla tietoturvallista ja riittävästi salattua. Lähetettävä data ei saa sisältää käyttäjää yksilöiviä tietoja. Komponentin data lähetetään HTTPS-protokollalla, joten palvelimen ja järjestelmän välinen yhteys on aina salattua. Yksilöiviä tietoja ei

myöskään kannata lähettää sellaisenaan, vaan yksilöivät tiedot satunnaistetaan asiakkaan puolella siten, että ne pystytään palauttamaan vain palvelimen päässä. Satunnaistettujen tunnisteiden avulla tiedot pidetään anonyyminä, joka suojaa lähetettävän tiedon mahdollisilta verkkokaappauksilta. Yleisesti ottaen yksittäinen käyttäjä ei ole relevanttia metriikkadatassa, mutta käyttäjien trendit pystytään tunnistamaan, mikäli samalta käyttäjältä tulevat tiedot pystytään ryhmittelemään tunnisteiden avulla.

Instrumentoinnin suorituskyky

Suorituskyvyn monitorointi ei saa olla liian raskas. Monitorointi toteutetaan instrumentointiperiaatteella, jolloin mittaustarkkuus pysyy hyvänä, mutta suoritus-kustannuksia saattaa aiheutua. Monitoroinnin suunnittelussa ja käyttöönotossa pitää ottaa huomioon keräilijöiden sijainnit ohjelmakoodissa. Metodit, joita kutsutaan silmukoiden sisällä on syytä jättää monitoroimatta, mikäli suorituskustannuksista tulee ongelma.

On huomattava, että instrumentoinnissa keräilijä suorittaa ylimääräistä koodia. Tästä syystä lisätyn toiminnallisuuden ajaminen vaikuttaa suoraan mitattuun arvoon, vaikkakin minimaalisesti. Näin olleen mittaussysteemi ei ole ideaalinen. Heisenbergin epätarkkuusperiaatteen mukaisesti ohjelman suorituskykyä ei pystytä mittaamaan äärettömän tarkasti, jos mittauksen toteutuksessa suoritetaan ylimääräistä ohjelmakoodia (Joyce 1967).

4.5 Soveltuvan metriikkakirjaston valinta

Työssä käytettävän metriikkakirjaston valintaan käytettyjä kriteereitä ovat palveluiden helppokäyttöisyys, palveluiden tarjoamat ominaisuudet ja niiden soveltuvuus työssä tehtävään suorituskykymonitorointiin.

Metrics .Net tarjoaa useita valmiita metriikoita datan keräämistä varten, jopa enemmän kuin Microsoftin Application Insights -palvelu. Application Insights ei tarjoa suoraa tilatarkastuspalvelua, toisin kuin Metrics .Net. Tilatarkastusten itse toteuttaminen monitorointikomponenttiin ei kuitenkaan vaadi suurta työtä.

Datan lähettämisessä ei kirjastojen välillä ole eroja, sillä molemmat toimivat automaattisesti lähettämällä tietyn väliajoin dataa eteenpäin. Palveluiden vastaanottavan pään (back-end) käyttöönotossa on kuitenkin suuria eroja, sillä Application Insights toimii suoraan Azure-palveluiden kautta ja palvelun asentaminen on yksinkertaista. Metrics .Net tarjoaa monia liitännäisiä vastaanottavan pään toteuttamista varten, mutta vaatii ylimääräistä työtä datan keräystä ja tallennusta varten. Vastaanottavan palvelinpään rakentaminen on työläämpää kuin Azure-palveluiden käyttöönotto.

Koska asiakasyritys on myös mukana Microsoft Partner -ohjelmassa, Microsoftin omien palveluiden käyttö on myös järkevää. Kaikki ohjelmistojen metriikat ja suorituskykydata pystytään ylläpitämään Azure-palvelussa ja käyttämään dataa jatkoanalysointia varten

ilman tiedon siirtämistä palvelusta toiseen. Application Insights tarjoaa myös valmiita työkaluja data-analyysiin ja datan visualisointiin. Metrics .Net -kirjastoa käytettäessä muut datan hallintaan käytettävät liitännäiset ja järjestelmät pitäisi manuaalisesti konfiguroida kehittäjäyrityksen toimesta. Tästä syystä tässä työssä käytetään Application Insights -analytiikkapalvelua.

4.6 Aspektikehys ja PostSharp

Ohjelmiston suorituskyvyn mittaus on hyvä toteuttaa siten, että komponentin käyttöönotto on helppoa ja vaatii mahdollisimman vähän muutoksia valmiiseen koodipohjaan. Tästä syystä komponenttiin tarvitaan aspektikehys, jonka avulla aspektiohjelmointia voidaan hyötykäyttää metriikoiden keräämisessä. Tässä työssä käytetään PostSharp-kirjaston (PostSharp 2015) aspektikehystä, joka on yksi metodikeskeytyksiä tarjoavista toteutuksista C#-kielelle.

PostSharp on C# ja Visual Basic (Ridgeway 2008) -ohjelmointikielille suunnattu apukirjasto, jonka tarkoituksena on auttaa kehittäjiä toimittamaan ominaisuuksia nopeammin ja pienentämään virheiden määrää. Kirjasto tarjoaa suunnittelumalleja, joiden avulla koodipohjan abstraktiotasoa voidaan nostaa ja vähentää koodiklooneja järjestelmässä. Se laajentaa kieliä C# ja Visual Basic tarjoamalla säieturvallisuutta kirjastoa käyttäviin järjestelmiin. Suunnittelumallien tarkoitus on vähentää koodin määrää usein käytetyistä toiminnoista, kuten säikeistyksestä, ominaisuuksien muutoksien seurannasta ja virhetietojen hallinnasta. Kirjasto tarjoaa valmiita toteutuksia yleisimpiin malleihin ja antaa työkalut omien mallien luomiseen. Kokonaisuutena PostSharp nostaa kehittäjien työn abstraktiotasoa, jotta he voivat keskittyä tärkeämpään liiketoimintalogiikan kehittämiseen. (PostSharp 2015)

Tässä työssä PostSharp -laajennuksesta käytetään vain siihen luotua aspektikehystä. Tämä on aspektikonseptin käytännönläheinen toteutus, jonka avulla monikäyttöiset toiminnallisuudet voidaan sisällyttää luokkiin, metodeihin, kenttiin, ominaisuuksiin tai tapahtumiin. Kirjaston tarjoamat metodikeskeytykset tarjoavat periytettävät kantaluokat, joissa liitoskohdat on lisätty valmiiksi. Liitoskohtien käyttäminen aspekteissa on mahdollista ylikirjoittamalla (override) virtuaalinen OnInvoke-metodi. Tämän metodin sisälle voidaan lisätä aspektiohjeita, joita halutaan suorittaa keskeytetyn metodin ympärillä. PostSharp linkittää aspektien toiminnallisuudet ohjelmaan käännösaikana eli käyttää käännösaikaista aspektipunontaa. Aspektikehystä käyttämällä voidaan monitoroitavan järjestelmän liiketoimintalogiikkaa ohjelmoivilta kehittäjiltä piilottaa monitorointiin liittyvät ominaisuudet. Kehittäjien ei tarvitse sekaantua monitoroinnin logiikkaan, vaan voivat keskittyä täysin järjestelmän logiikan ohjelmointiin.

4.7 Monitorointiprosessi

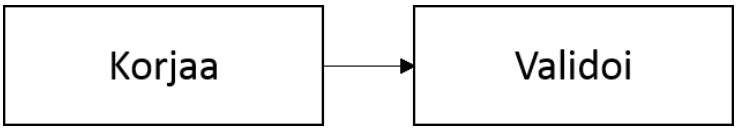
Monitorointiprosessi jakautuu viiteen eri vaiheeseen. Tässä luvussa tarkastellaan prosessin kulkua kehittäjien näkökulmasta ja miten komponentti auttaa yrityksen monitorointiprosessissa. Prosessin alkuvaiheita kehittäjän näkökulmasta voidaan kuvata kuvassa 6 esitetyllä kaaviolla.

Kuva 6. *Monitorointiprosessin alkutilanne.*

Kerätyllä datalla pyritään tunnistamaan ongelmat järjestelmässä. Tämän jälkeen kehittäjien tulee luokitella ja lajitella ongelmat. Tässä tulee ottaa huomioon ongelman laajuus ja se, kuinka moneen käyttäjään ongelma kohdistuu tai kuinka usein ongelma ilmenee. Diagnosoinnin tarkoituksena on löytää ongelman aiheuttaja ja ongelman kohdistaminen tiettyyn osaan järjestelmässä.

Komponentin avulla kerättäviä metriikoita ja virhelokeja apuna käyttäen ongelmia pystytään tunnistamaan ja lajittelemaam. Komponentin keräämät tiedot voidaan jo keräysvaiheessa lajitella mittauskohteen tarkoituksen perusteella. Tämän avulla käsittely on jatkossa helpompaa. Diagnosointi pystytään toteuttamaan joko lajittelemalla kerättyjä tietoja tarkemmin tai diagnosoinnin apuna käytetään komponentin jaottelua.

Kun ongelmat ovat diagnosoitu, kehittäjän tulee reagoida ongelmaan ja tutkia parannusvaihtoehtoja. Kuvassa 7 on kuvattu diagnosoinnin jälkeinen prosessi. Ongelman diagnosoinnin yhteydessä ongelma on kohdistettu tiettyihin kohtiin järjestelmässä. Korjausvaiheessa ongelman aiheuttaneeseen kohtaan tehdään korjaus. Ongelman korjauksessa on huomioitava ohjelmayksikön suhteet muihin yksiköihin ja tarkastaa, että korjaus ei muuta liittyvien ohjelmayksiköiden käyttöä ja toimintaa. Korjausvaiheen jälkeen on tehtävä validointi ja tarkastettava, että koko ongelma on saatu korjattua.



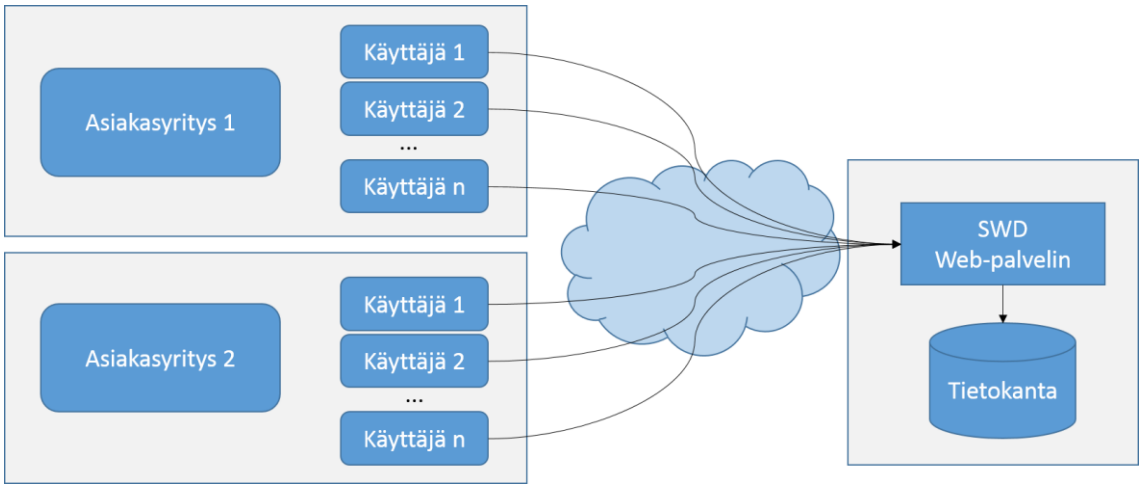
Kuva 7. *Diagnosoinnin jälkeinen prosessi*

Monitorointikomponentin on tarkoitus pystyä vastaamaan tähän kehittäjien prosessiin siten, että ensimmäinen kuvattu prosessi ongelman diagnosointiin asti pystyttäisiin tekemään monitorointikomponentin avulla lähetetyn datan avulla. Tämän vuoksi

lähetettävän datan on oltava tarpeeksi tarkkaa ja informatiivista, jotta ongelma pystytään kohdistamaan mahdollisimman helposti. Virhetilanteissa myös poikkeustietojen lähettäminen on olennaista, jotta ongelmat tunnistetaan ja kohdistetaan helposti.

5. SUORITUSKYKYMONITOROINNIN TOTEUTUS

Monitorointikomponentti kerää tietoa ohjelman toiminnasta muistiin ja lähettää sen verkon yli Microsoftin Azure -palvelimille. Komponentti ei ota kantaa asiakasyrityksen verkkoinfrastruktuuriin, vaan lähettää monitorointidatan suoraan sovellusta suorittavalta koneelta. Toinen mahdollisuus olisi koota tiedot järjestelmän tietokantaan ja lähettää tiedot tietokantapalvelimelta SWD:n palvelimille. Suurissa teollisuusyrityksissä tietokantapalvelimet sijaitsevat suojatummissa virtuaaliympäristöissä, joista ei välttämättä ole pääsyä ulkoverkkoon. Tästä syystä HTTP- ja HTTPS-protokollat mahdollistavat varmemman lähetysmekanismin myös suurten yritysten monitorointia silmällä pitäen. Kuvassa 8 kuvataan asiakkaiden SWD PES-järjestelmän käyttäjien ja monitorointidatan keräävän web-palvelimen yleisjärjestely.



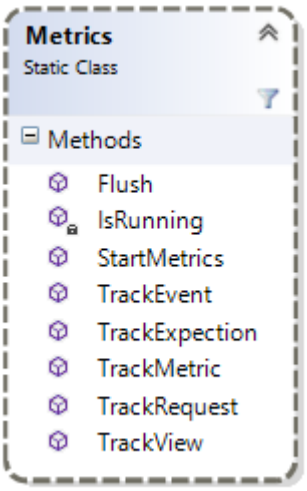
Kuva 8. *Monitorointikomponentin datan lähetys tuotantoympäristöstä SWD:n palvelimelle.*

Asiakasyrityksillä on vaihteleva määrä käyttäjiä ja jokainen monitoroinnin piirissä oleva käyttäjä lähettää erikseen dataa web-palvelimelle, jossa tiedot kerätään ryhmitellen jatkoprosessointia varten ja siirretään tietokantaan. Tietokantaan tallennettua metriikkadataa voidaan analysoida ja koostaa käytettävämpään muotoon. Tietokantaan siirtäminen toteutetaan mahdollisesti jatkossa Application Insightsin jatkuvan vientitoiminnon (continuous export) avulla (Wills 2015).

5.1 Metriikkaluokka

Metriikkaluokka on rajapinta käytettyyn metriikkakirjastoon. Luokka on staattinen, minkä seurauksena metriikkaluokan metodeja pystytään kutsumaan kaikissa ohjelman

kohdissa, joihin monitorointikomponentin nimiavaruus on lisätty käytettäväksi. Kuvassa 9 on esitetty metriikkaluokan UML-luokkakaavio.



Kuva 9. *Metriikkaluokan UML-luokkakaavio.*

Metriikkaluokka toimii kääreluokkana, ja sen tarkoitus on välittää ja kääntää metodikutsut monitoroinnissa käytettävän metriikkakirjaston kutsuiksi. Tämän tarkoitus on luoda modularisoitu ratkaisu metriikkadatan keräämiseen. Mikäli alla oleva metriikkakonteksti halutaan vaihtaa, kehittäjän ei tarvitse kuin toteuttaa näiden ulospäin näkyvien metodien kääntäminen käytetyn metriikkakontekstin omiksi kutsuiksi.

Metriikkaluokan tarjoamia metodeja ovat:

StartMetrics alustaa metriikkaluokan, jolloin tämä saa metriikkakirjaston instrumentaatioavaimen palvelinyhteyttä varten ja se sidotaan käytettävään metriikkakontekstiin palvelimen puolella. SWD PES:iin tehtävä monitorointi voidaan toteuttaa yhtä instrumentaatioavainta käyttäen.

TrackMetric on tarkoitettu yksittäisten metriikka-arvojen lähettämiseen. Metodille annetaan parametriksi metriikan nimi, arvo ja mahdolliset lisämääreet, joiden avulla metriikkaa voidaan luokitella jatkokäsittelyn yhteydessä. Metriikan nimi on merkkijono, jonka pitää olla yksilöllinen tietylle metriikalle, jotta metriikka-arvot eivät sekoitu keskenään arvoja käsitellessä. Kerättävän metriikan arvo on numeerinen. Alla yhden datapisteen kerääminen metriikkadataan:

```
double sum = a + b;
Metrics.TrackMetric(”sum”, sum, null);
```

TrackRequest on tarkoitettu ohjelmassa suoritettujen kutsujen seurantaan. Metodin tarkoituksena on kerätä kutsutun metodin suoritukseen liittyvää dataa. Metodin kesto, metodin onnistuminen ja palvelun vastaus voidaan lähettää metriikkadataan tämän avulla.

Metodin parametreina annetaan kerättävän metriikan nimi, joka kutsujen seurannan tapauksessa kannattaa olla kutsutun metodin nimi tai muu yksilöllinen tunniste, jonka avulla metodi voidaan luokitella jatkokäsittelyssä. Parametrina annetaan myös kutsun kesto sekunteina. Tieto kutsun vastauksesta on tarkoitettu lähinnä ulkopuolisiin palveluihin, jotka antavat erillisen vastauksen. Ohjelman sisäisten kutsujen vastauksia ei ole välttämätöntä lisätä kerättävään dataan, koska sisäiset kutsut eivät välttämättä palauta erillista palvelun onnistumiseen liittyvää vastausta. Kutsun onnistuminen voidaan antaa halutessa metriikkametodille. Oletusarvoisesti metodi olettaa kutsun onnistuneen. Tällekin metodille voidaan antaa haluttuja lisämääreitä kerättäväksi kutsumetriikan lisäksi. Oheisessa esimerkissä on yksinkertaisesti ajastettu yksittäisen laskentametodin suoritusaika ja lähetetty tämä metriikkadataan.

```
Stopwatch watch = Stopwatch.StartNew();
DateTime startTime = DateTime.Now;
DoCalculations();
watch.Stop();

Metrics.TrackRequest("DoCalculations", startTime, watch.Elapsed);
```

TrackEvent auttaa seuraamaan yksittäisiä ohjelmassa suoritettavia tapahtumia. Parametriksi sille annetaan suoritetun tapahtuman nimi. Mahdollisina lisäparametreina metodille voidaan antaa myös ylimääräisiä lisämääreitä ja metriikka-arvoja. Tämän metodin avulla ei kuitenkaan ole pääasiallisesti tarkoitus kerätä numeerisia arvoja. Se soveltuu tapauksiin, joissa numeerisista arvoista ei saada hyötyä, vaan halutaan laskea tapahtuvien toimintojen määrää tai seurata tapahtumien suoritusjärjestystä.

Jos tutkittavat tapahtumat luokitellaan lisämääreiden avulla sopivasti, voidaan tämän metodin avulla saada kerättyä ohjelman suorituksen aikaisia tapahtumaketjuja. Näin voidaan varmistaa, että tutkittavat asiat suoritetaan aina oikeassa järjestyksessä. Alla olevassa esimerkissä on kerätty tieto tietyn käyttöliittymäpainikkeen painalluksesta.

```
private void buttonCreateEvent_Click(object sender, EventArgs e)
{
      Dictionary<string, string> properties = new Dictionary<string,string>();
      properties["Method"] = MethodBase.GetCurrentMethod().Name;

      Metrics.TrackEvent("buttonEvent", properties);
}
```

TrackException on tarkoitettu poikkeustapahtumatietojen keräämiseen. Ohjelman poikkeustilanteista saadaan kerättyä tämän metodin avulla antamalla parametriksi käytetty poikkeusolio. Tämän lisäksi metodille voidaan antaa lisämääreitä ja kerättäviä metriikka-arvoja parametriksi poikkeustietojen ryhmittelyä varten. Oheisessa esimerkissä (Ohjelma 1) otetaan kiinni poikkeustilanne ja kerätään tiedot metriikkadataan. Lisäksi virheen aiheuttavan metodin nimi laitetaan lisämääreenä kerättävään dataan.

```
try
{
    object obj = null;
    obj.ToString();
}
catch (NullReferenceException ex)
{
    Dictionary<string, string> properties =
        new Dictionary<string, string>();
    properties["Method"] = MethodBase.GetCurrentMethod().Name;

    Metrics.TrackExpection(ex, properties);
    throw ex;
}
```

Ohjelma 1. *Poikkeustilanteen tietojen kerääminen.*

TrackDependency kartoittaa ulkoisiin palveluihin tehtyjä kutsuja. Metodin tarkoitus on toimia suurilta osin samoin kuin kutsujen seurantaan tarkoitetun TrackRequest-metodin. Se ottaa parametrikseen ulkoisen riippuvuuden eli tässä tapauksessa palvelun nimen, palvelun käskyn nimen, suorituksen alkuajan, suorituksen keston ja onnistumisinformaation.

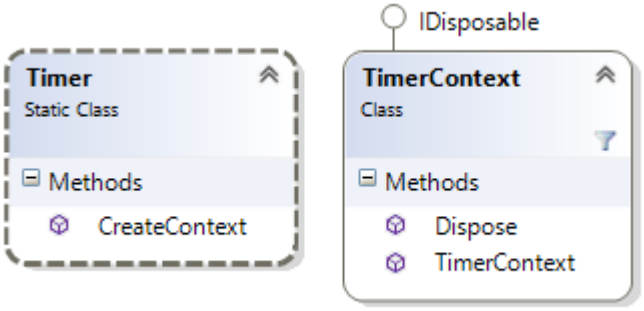
Jos samanlainen datankeräys on käytössä myös ulkoisessa palvelussa, pystytään näiden toimintojen avulla seuraamaan palvelukutsujen aiheuttama kokonainen palveluketju. Jos ulkoisessa palveluissa myös mitataan ulkoisen palvelujen sisäisten toimintojen kesto, pystytään myös tarkastelemaan koko palveluketjun suorituskykyä. Tätä toimintoa voidaan käyttää esimerkiksi SWD PES -järjestelmän käyttämissä mikropalveluissa (micro services), joissa palvelu suoritetaan eri palvelimella tai ainakin eri prosessissa kuin järjestelmäinstanssi. Tällaisia mikropalveluita voisivat olla esimerkiksi varasto- tai materiaalitarvelaskenta, jotka voidaan suorittaa erillisenä prosessina SWD PES -sovelluksen ulkopuolella.

TrackView kerää tietoa näkymien vaihdoista ohjelmiston sisällä. Metodille annetaan parametriksi näkymän nimi. Tämän toiminnon avulla pystytään tutkimaan käyttäjän käyttäytymistä ja navigointia ohjelman sisällä.

Flush lähettää edelliseen lähetykseen asti kerätyt datat palvelimelle. Metriikkapalvelu lähettää automaattisesti dataa sopivin väliajoin, mutta ei tee sitä esimerkiksi ohjelmaa lopetettaessa. Tätä metodia lopuksi kutsumalla myös ohjelman lopetukseen asti kerätyt datat saadaan lähetettyä palvelimelle.

5.2 Ajastin

Ajastinluokan on tarkoitus toimia sekuntikellona aspekteissa tehtävissä kellotus-operaatioissa. Ajastimen julkinen rajapinta on toteutettu minimaalisesti, jotta ajastinta ei käytetä väärin. Kuvassa 10 on esitelty ajastimen ja ajastinkontekstin UML-luokkakaavio.



Kuva 10. *Ajastimen UML-luokkakaavio*

Timer-luokka on staattinen, minkä vuoksi sitä voidaan kutsua mistä tahansa ohjelman suorituskohdasta. *Timer*-luokka luo ajastinkontekstin, joka toimii sekuntikellona. Kun ajastinkonteksti ollaan tuhoamassa, sen mittaama aika luonnista tuhoamiseen asti otetaan talteen ja kutsutaan luonnissa parametrina annettua lopetustoimintoa:

CreateContext luo uuden ajastinkontekstin. Kontekstin luonnin yhteydessä sekuntikello laitetaan päälle. Tälle metodille annetaan parametriksi toiminto, jota kutsutaan ajastinkontekstin tuhoamisen yhteydessä.

TimerContext alustaa sekuntikellon ajastinkontekstiin. Kelloa käytetään ajan mittaamiseen, kunnes konteksti tuhotaan. Kontekstin rakentajalle annetaan myös toiminto, joka suoritetaan, kun konteksti tuhotaan. Tälle toiminnolle annetaan kulunut aika parametriksi.

Dispose tekee ajastinkontekstin tuhoamiseen liittyvät toimenpiteet. Tätä metodia kutsutaan automaattisesti, mikäli koko ajastinkontekstin käyttö on sijoitettu using-lausekkeen sisälle ohjelmassa. Tämä metodi sammuttaa päällä olleen sekuntikellon ja kutsuu rakentajalle parametriksi annettua toimintoa ja antaa toiminnolle parametriksi kuluneen ajan *TimeSpan*-oliossa.

Ohessa on esimerkki ajastimen käytöstä using-lausekkeen avulla.

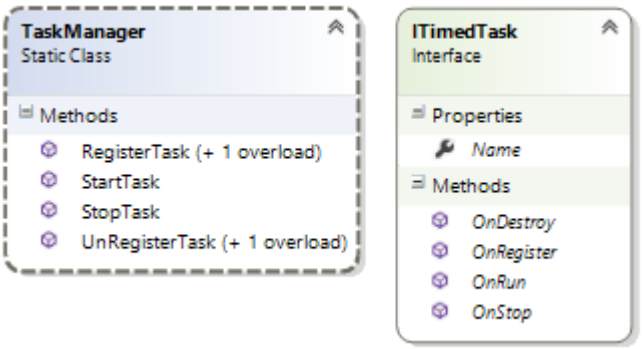
```
using (Timer.CreateContext("contextName", (duration) =>
      AfterTiming(duration)))
{
      DoCalculations();
}
```

Using-lausekkeen käyttäminen varmistaa ajastimen oikean käytön ja sen, että se tuhotaan ajastusta lopettaessa. Ajastinkonteksteja voidaan käyttää sisäkkäin siten, että osa mitattavasta toiminnasta voidaan sijoittaa ulomman ajastimen alueelle, mikäli halutaan mitata yksittäistä toimintoa tarkemmin jo valmiiksi päällä olevan ajastimen lisäksi.

5.3 Ajoitetut tehtävät

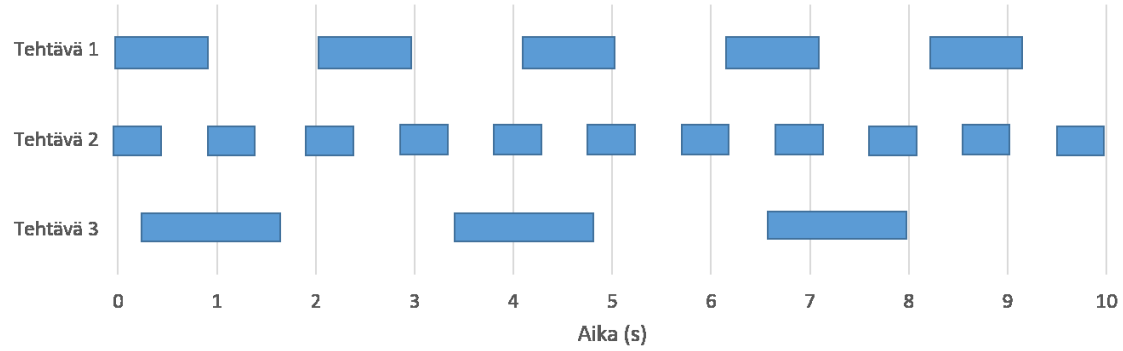
Komponentin ajoitetut tehtävät mahdollistavat myös profilointityyppisen monitoroinnin suorittamisen komponentin avulla. Järjestelmän käynnistyksen yhteydessä komponenttiin voidaan rekisteröidä ajoitettuja tehtäviä, joita suoritetaan säikeistettynä muun järjestelmän suorituksen ohella. Rekisteröinti tehdään komponentin *TaskManager*-luokkaan. Tehtävänhallintaluokkaan voidaan rekisteröidä ja poistaa tehtäviä myös suorituksen aikana jälkikäteen, mikäli osa-aikainen mittaaminen on haluttua.

Ajoitettujen tehtävien tarkoitus on mitata määritetyin aikavälein asoita järjestelmän toiminnassa. Ladattujen tietorakenteiden kokoja tai muita muistinkäyttöön liittyvää tietoa voidaan kerätä komponenttien sisällä koko järjestelmän suorituksen ajalta. Profilointitehtävät toteutetaan asiakasprojekteissa räätälöitynä. Tehtävän tulee toteuttaa annettu rajapinta, joka on määritelty monitorointikomponentissa. Kuvassa 11 on esitetty tehtävämanagerin rakenne UML-luokkakaaviona.



Kuva 11. *Tehtävämanagerin ja ajoitetun tehtävän UML-luokkakaavio.*

Tehtävähallinnan tarkoitus on ylläpitää kaikkia profiloivia tehtäviä, joita komponenttiin on rekisteröity. Ajoitetut tehtävät rekisteröidään yksilöivän avaimen avulla. Tämän avulla tehtäviä pystytään myös pysäyttämään ja poistamaan tehtävähallinnasta. Avaimen lisäksi rekisteröinnin yhteydessä määritetään jakso, kuinka usein tehtävä yritetään suorittaa. Jos tehtävä on suorituksessa, kun se halutaan uudelleenkäynnistää, tehtävänhallinta ohittaa suorituskierroksen. Tämän jälkeen tehtävää yritetään suorittaa vasta seuraavan aikajakson kuluttua. Kuvassa 12 on esitetty esimerkki tehtävänhallinnan aikataulutuksesta.



Kuva 12. *Tehtävienhallinnan suoritusaikataulu.*

Ajoitetulle tehtäville kutsutaan sen metodeita tarvittavien ehtojen mukaisesti. Tehtävän OnRun-metodia kutsutaan silloin, kun tehtävä halutaan suorittaa. Alustustoimenpiteitä voi tehdä OnRegister-metodin suorituksen yhteydessä. Kun tehtävä poistetaan tehtävänhallinnasta OnDestroy-metodin avulla voidaan tehdä tarvittavat siivous-toimenpiteet ja vapauttaa mahdollisesti tehtävässä varattu muisti ja viitteet muihin olioihin. OnStop-metodin tarkoitus on pysäyttää tehtävä, mikäli tehtävä on haluttu pysäyttää välittömästi eikä suoritusta haluta enää jatkettavan.

Ajoitettujen tehtävien tarkoituksena on tehdä pieniä mittauksia ohjelman suorituksen aikana. Sen ei ole tarkoitus tehdä suurta laskentaa tai mitata tehtävän aikana pidemmältä aikaväliltä asioita. Näin ajoitetut tehtävät saadaan pidettyä minimaalisena, eikä suorituskykykustannuksia aiheudu profiloinnin aikana. Tehtävien käytössä ja luomisessa järjestelmän mittaamisessa vastuu on järjestelmän kehittäjällä ja näin ollen vaatii huolellista suunnittelua ennen tehtävien rekisteröintiä tuotantoympäristössä suori-tettavaan monitorointiin.

5.4 Datan kerääminen

Monitorointidatan kerääminen toteutetaan luvussa 4.3.2 esitellyn Microsoft Application Insights -metriikkakirjaston avulla. Application Insights tarjoaa telemetriakomponentin, jonka avulla dataa kerätään muistiin väliaikaisesti.

Datan keräämiseen käytetään apuna esiteltyä aspektikehystä, josta löytyy mittaukseen tarvittavat aspektitoteutukset. Komponentin suorittama tiedonkeräämisen hallinta toteutetaan aspektiattribuuttien avulla. PostSharp-kirjaston pohjalta luotujen aspektien avulla mitattavan sovelluksen metodeja ja luokkia voidaan koristella halutuilla aspektiattribuuteilla, mikä aktivoi kohteet monitoroinnin piiriin. Aspektit periytyvät C#-attribuuteista (Freeman 2010), mikä mahdollistaa niiden käyttämisen oheisen esimerkin tavoin metodien koristeluun. Koska aspekteja käytetään attribuuttien avulla, valmiiseen koodipohjaan metodien sisällä ei tarvitse koskea ja näin pienennetään riskiä, että aspektit

muuttaisivat sovelluksen toimintaa. Oheisessa attribuuttiesimerkissa metodiin lisätään suoritusajan mittausaspekti.

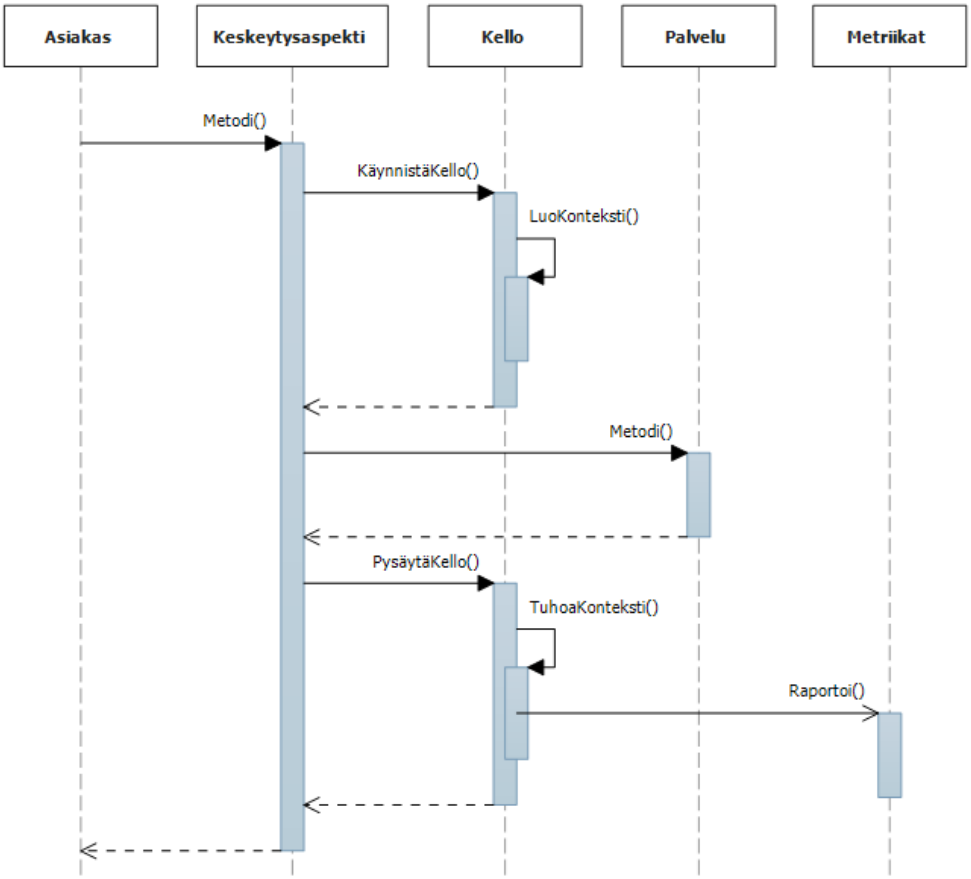
```
[Aspects.TrackRequest]
void DoCalculations()
{
}
```

Suorituskykymonitorointi metoditasolla toteutetaan TrackRequest-aspektilla, joka tekee liitoskohdat metodin suorituksen ympärille. Ohjelman käännöksen yhteydessä metodin ympärille liitetään aspektin ohjeet määriteltyyn liitoskohtaan. Aspekti periytyy *MethodInterceptionAspect*-luokasta, joka tarjoaa virtuaalisen toteutuksen OnInvoke-metodista. Tämä suoritetaan, kun alkuperäinen metodi (DoCalculations) halutaan suorittaa. Oheisessa koodiesimerkissä (Ohjelma 2) on näytetty TrackRequest-aspektin toteutus.

```
[Serializable]
public class TrackRequestAttribute : MethodInterceptionAspect
{
    public override void OnInvoke(MethodInterceptionArgs args)
    {
        Action<DateTimeOffset, TimeSpan> postTraceAction =
        new Action<DateTimeOffset, TimeSpan>((startTime, duration) =>
            {
                AfterTrace(args, startTime, duration);
            });

        using (Timer.CreateContext(string.Format("{0} - TrackRequest",
               args.Method.Name), postTraceAction))
        {
            base.OnInvoke(args);
        }
    }

    protected virtual void AfterTrace(MethodInterceptionArgs args,
        DateTimeOffset startTime, TimeSpan duration)
    {
        string response = args.ReturnValue != null ?
            args.ReturnValue.ToString() : "NO_RESPONSE"

        Metrics.TrackRequest(args.Method.Name, startTime,
            duration, response, true);
    }
}
```

Ohjelma 2. *TrackRequest-aspektin rakenne.*

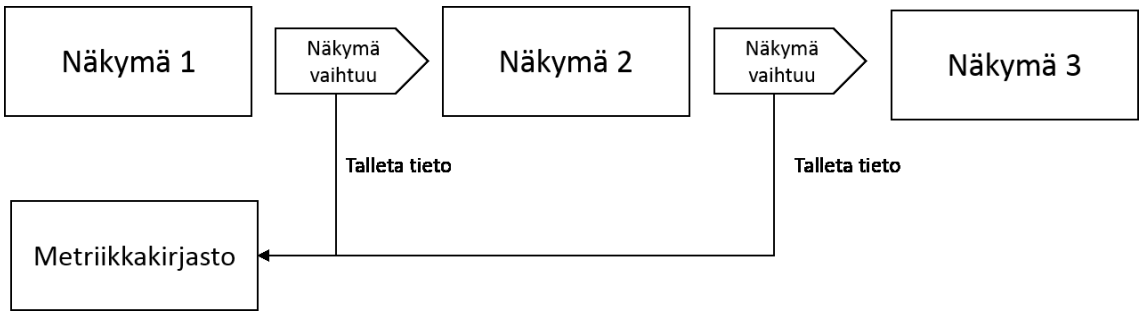
Tällaisen metodikeskeytyksen avulla pystytään lisäämään instrumentoinnin tarvitsemaa toiminnallisuutta suoritettavaksi ennen halutun metodin suoritusta ja suorituksen jälkeen. Kuvassa 13 on kuvattu metodikeskeytyksen avulla toteutetun suoritusajan instrumentointiaspektin toiminta UML-sekvenssikaaviona.



Kuva 13. *Keskeytysaspektin avulla tehdyn kellotuksen UML-sekvenssikaavio.*

Suorituksen keskeyttävän aspektin avulla pystytään mittamaan metodin suoritusaika tehokkaasti ja ilman muutoksia suoritettavaan metodiin. Kellotusaspektin ajastimena käytetään komponenttiin luotua ajastinluokkaa, joka toimii oman näkyvyysalueensa alueella ja pysäyttää kellon alueelta poistuessa. Pysäytyksen jälkeen kellon mittaama aikatieto kerätään talteen lähetystä varten. Toiminnallisuus on uudelleen käytettävissä missä tahansa metodissa ilman muutoksia metodin koodiin, koska se toteutetaan aspektin avulla.

Monitorointikomponenttiin luotavan aspektikehyksen avulla pystytään myös toteuttamaan toimintoja, joissa ei tehdä mittauksia. Järjestelmän laadun parantamiseksi komponenttiin halutaan lisättäväksi myös käyttäjän käyttöliittymänavigoinnin seurantaa. Tämän avulla käyttäjien näkymien vaihdot järjestelmässä pystytään analysoimaan ja analyysista voidaan nähdä yleiset navigointipolut järjestelmässä. Usein käytettyjen navigointipolkujen analysoinnilla voidaan parantaa järjestelmän valikkorakennetta, jotta usein käytetyt toiminnot ovat helposti käyttäjän saatavilla. Navigoinninseuranta on kuvattu kuvassa 14.



Kuva 14. *Navigoinninseurannan yleiskuvaus*

Jotta navigoinnin seuranta ei ole riippuvainen SWD PES -alustasta, lisätään monitorointi-komponenttiin perustoteutus navigoinninseurannan toteuttavalle aspektille. Tämä vaatii sen, että SWD PES -alustan perusnäkymän, joista kaikki järjestelmän näkymät tulisi periyttää, tulee toteuttaa navigoinninseurantaan vaadittavat tiedot sisältävän rajapinnan. Tämä vaatii myös pieniä muutoksia peruskomponenttien koodiin alustan puolella.

5.5 Datan lähettäminen

Application Insights kerää datan muistiin väliaikaisesti, jotta verkkoa ei kuormiteta lähetettävän datan vuoksi. Dataa saattaa kertyä satoja datapisteitä sekunnissa, joten datan lähettäminen välittömästi ei ole mielekästä.

Application Insights yrittää valita datan lähettämiseen ajankohdan, jossa vaikutus loppukäyttäjälle olisi mahdollisimman pieni. Muistiin kerätty data lähetetään Microsoftin Azure-palveluun suoraan HTTPS-protokollan avulla. (Wills 2015)

Lähetettävän datan määrittely

Monitorointidatan yhteneväisyys ja laajuus parantavat datan jatkokäsittelyä ja analysointia. Data on järkevää pitää tiiviinä, mutta silti kaiken olennaisen sisältävänä, koska monitoroinnin keräämät mittaustiedot lähetetään verkon yli.

Application Insights kerää ja lähettää dataa hyvin rakenteellisena. Lähetettävä data on JSON muodossa (Bryan & Zyp 2012), josta kaikki tarvittavat tiedot ovat periaatteessa myös ihmisen luettavissa. Kehitysympäristössä lähetettävä data pystytään tulostamaan myös kehityskonsoliin, jonka avulla koodin toiminnan ja kerättävien tietojen tarkastelu on mahdollista. Ohessa on esimerkki (Ohjelma 3) lähetettävästä JSON-datasta.

```
Application Insights Telemetry:
{
    "name":"Microsoft.ApplicationInsights.
    xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    .Request",
    "time":"2015-10-05T13:03:15.4035052+03:00",
    "iKey":"xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
    "tags":{"ai.internal.sdkVersion":"1.2.0.5639",
    "ai.device.osVersion":"Microsoft Windows NT 6.2.9200.0"},
    "data":
    {
        "baseType":"RequestData",
        "baseData":
        {
            "ver":2,
            "id":"2119148598780947428",
            "name":"testMethod",
            "startTime":"2015-10-05T13:03:15.4035052+03:00",
            "duration":"00:00:00.2031370",
            "success":true,
            "responseCode":"Success"
        }
    }
}
```

Ohjelma 3. *Esimerkki Application Insights -palveluun lähetettävästä JSON-datasta.*

Ensimmäisessä datalohkossa on metriikkadatan perustiedot, instrumentaatioavain ja palveluun lisätyt versiotunnisteet. Seuraava datalohko määrittelee kerätyn datan tyypin ja mahdolliset lisämääreet. Sisimmäinen datalohko sisältää kerätyn datan, jossa on metriikan tunniste, nimi ja esimerkin tapauksessa testatun metodin kesto ja aloitusaika.

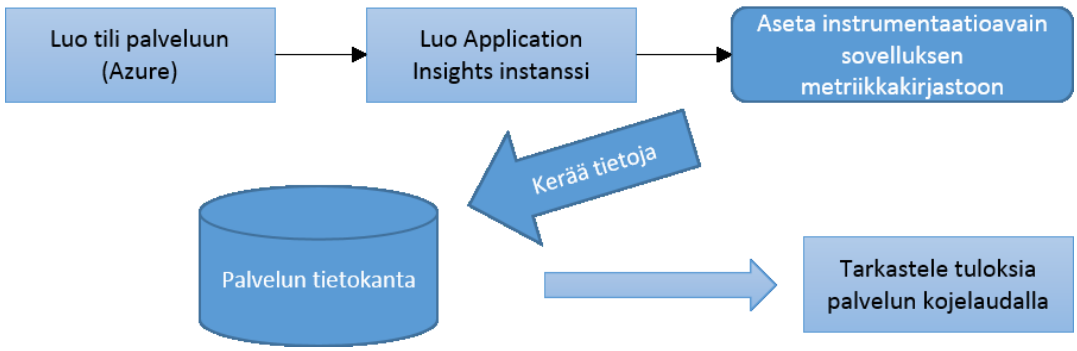
Tietoturva ja salaus

Kaikki Application Insightsin lähettämä data on salattua. Application Insights lähettää kaiken datan Azureen HTTPS:n kautta. Kerätty data tallennetaan Azuren palvelimille Yhdysvaltoihin, joten tietokantapalvelimien sijainnin kautta muodostuviin tietoturvauhkiin työssä ei pystytä ottamaan kantaa.

Kerättävässä monitorointidatassa mahdollisesti liikkuvat käyttäjää yksilöivät muuttujat on myös mahdollista muuttaa pois selkokielisestä muodosta, jotta tuotantodataa tai asiakkaan tietoja ei lähetetä verkon yli ymmärrettävässä muodossa. Tähän voidaan käyttää kehittäjäyrityksen omia nimeämiskäytäntöjä, jolloin asiakkaalle määritellään tunniste, jonka avulla tunnistetaan tietty asiakas. Halutessaan tunnisteeseen voidaan käyttää jotain salausalgoritmia. Vastaanottamisen jälkeen tiedot voidaan yhdistää takaisin haluttuihin kokonaisuuksiin muuttamalla tiedot selkokielisiksi metriikka-analyysiä varten.

5.6 Datan tulkinta

Data-analyysiä ja tarkastelua voidaan tehdä Application Insights -palvelun sisällä. Kuvassa 15 on kuvattu Application Insights -palvelun käyttöönotto- ja keräysprosessi datan koostamista varten.

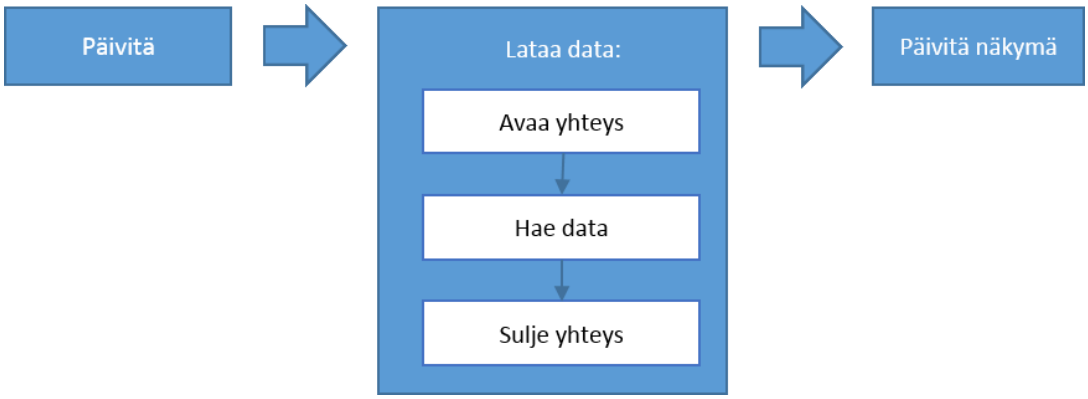


Kuva 15. *Application Insights -palvelun käyttönotto monitorointikomponenttiin.*

Application Insights -instanssin luomisen jälkeen kaikki samalla instrumentaatio-avaimella lähetetyt tiedot voidaan ryhmitellä omaksi datakokonaisuudeksi. Palvelun kojelaudalla (dashboard) voidaan tarkastella yksittäisiä metriikkatietoja tai koostaa laajempia kokonaisuuksia haluttujen kriteerien mukaan monien valittavien suodattimien avulla.

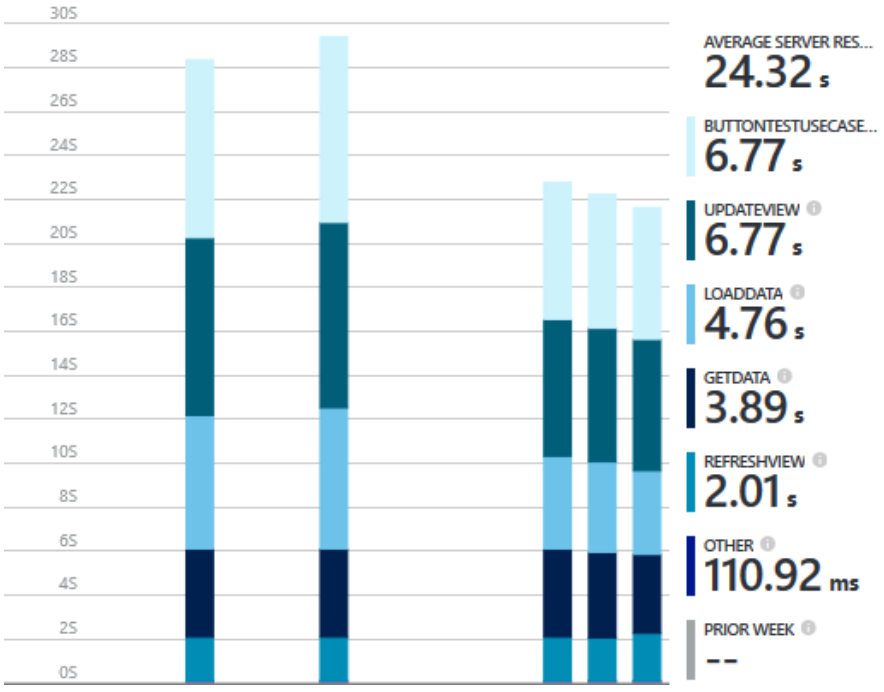
Näkymän päivitys

Suorituskykyn monitoroinnin testaamiseen on käytetty kuvassa 16 esitettyä käyttötapausta. Käyttötapaus on pelkistetty esimerkki näkymän lataamisesta ja päivittämisestä. Käyttötapauksen kaikki metodit on aktivoitu monitoroitavaksi TrackRequest-attribuutin avulla. Käyttötapaus alkaa käyttäjän syötteestä, jossa päivytyspainiketta on painettu. Datan latauksen jälkeen näkymä päivitetään.



Kuva 16. *Mitattavat operaatiot ja suoritusjärjestys näkymää päivitettäessä.*

Käyttötapauksen mittaamisen avulla pystytään tunnistamaan sen suorituspolun aiheuttamia suoritusaikoja ja mahdollisia ongelmakohtia. Monitorointitietojen lähettämisen jälkeen kehittäjä pystyy tutkimaan Application Insights -palvelun avulla käyttötapauskohtaisesti suoritusaikoja ja niiden jakautumista metodeittain. Käyttötapaukset voidaan myös luokitella käyttäjä- tai asiakaskohtaisesti. Kuvassa 17 näytetään palveluun kerätyn käyttötapauksen esimerkin keskimääräiset suoritusajat. Suoritusajat on ryhmitelty graafiin metodin nimen perusteella.



Kuva 17. *Näkymän päivitys -käyttötapauksen keskimääräiset suoritusajat ryhmiteltynä metodeittain Application Insights -palvelussa.*

Keskimääräinen aika tässä käyttötapausesimerkissä on noin 6,77 sekuntia. Kuvaajasta nähdään, että GetData-metodi on suoritusajaltaan suurin yksittäinen metodi mitatussa käyttötapauksessa. Keskimääräisesti 60 prosenttia suoritusajasta kuluu pelkästään datan hakemiseen. Tästä voidaan päätellä, että käyttötapauksen nopeuttamiseksi datan hakemista pitäisi optimoida. Optimointi tässä tapauksessa voisi tarkoittaa tietokantaan lähetettävän kyselyn tarkastamista, jotta kysely olisi nopein mahdollinen ja palauttaisi kannasta vain näkymän tarvitseman datan.

Toinen huomioitava toiminto on näkymän päivitys: 30 prosenttia ajasta kuluu näkymän päivitykseen. Tähän ei pitäisi kuulua enää datan hakemista muilta resursseilta. Tämä tarkoittaa, että suorittavan ympäristön ei pitäisi vaikuttaa merkittävästi tämän metodin suoritukseen. Tulosten saamisen jälkeen kehittäjä voi näin ollen tarkastella näkymän päivitystoimintoja jälleen kehitysympäristössä, jossa profilointityökaluilla saadaan tarkempia operaatiokohtaisia suoritusaikoja, joiden perusteella koko toiminnon nopeuttamista voidaan tehdä.

6. TULOKSET JA JOHTOPÄÄTÖKSET

Työn ja toteutetun komponentin suunnitteluvaiheessa komponentille määriteltiin vaatimuksia, joita toteutuksessa otettiin huomioon. Ohessa tarkastellaan näiden vaatimusten toteutumista. Komponentti toteutettiin C#- ja .Net-teknologioilla käyttäen apuna Microsoftin Application Insights -palvelua ja PostSharp-aspektikehystä. Komponentti sisältää kaksi osakokonaisuutta: metriikkarajapinnan tietojen keräämistä ja tallentamista varten ja aspektikehyksen instrumentointikeräilijöiden sijaintien määrittämiseen. Teknologiat soveltuivat toteutettuun komponenttiin ja komponentin käyttötarkoitukseen loistavasti. Valitut teknologiat tukivat osittain suoraan määriteltyjä vaatimuksia.

Aspektiohjelmoinnin käyttäminen työpöytäsovelluksen monitorointiin vaikuttaisi oivalta ratkaisulta, koska valmiin järjestelmän koodipohjaan tehtävät muutokset vaatisivat liikaa resursseja. Monitorointikoodien lisääminen valmiiseen ohjelmaan voisi myös lisätä virheitä järjestelmässä ja sotkisi koodia toiminnoilla, jotka eivät liity ohjelman varsinaiseen liiketoimintalogiikkaan.

Modulaarisuus

Modulaarisuus otettiin huomioon komponentin luokkajaottelussa. Luokkiin jako toteutettiin suoraan toimintojen perusteella siten, että luokka vastaa oman toimialueen operaatioista. Monitorointikomponentin metriikkarajapinta on toteutettu siten, että metriikkakonteksti pystytään vaihtamaan tarvittaessa käyttämään toista metriikkakirjastoa, mikäli nykyinen valinta ei jostain syystä ole riittävä kaikkiin käyttötapauksiin. Näin komponentti pystytään pitämään modulaarisena, eikä komponentin käytössä tarvitse sitoutua käyttämään pelkästään valittua Application Insights -metriikkarajapintaa.

Aspektikehyksenä käytetään PostSharp-kirjastoa. Tämäkin kirjasto pystytään korvaamaan uudella suhteellisen pienin muutoksin monitorointikomponenttiin. Aspektikehyksen on kyettävä toteuttamaan metodikeskeytykset ja muut tarvittavat toiminnallisuudet, jotta sitä voidaan käyttää. Kantaluokat toteuttamalla pystytään pitämään monitorointikomponenttiin toteutetut periytetyt aspektit samanlaisina, kunhan kantaluokat tarjoavat samat ominaisuudet. Aspektit myös jaettiin omiin toteutuksiinsa siten, että yksi aspekti vastaa yhdestä monitorointitehtävästä. Aspekteja voidaan lisätä kokoelmaan kun erilaisia monitorointitietoja halutaan kerätä. Näin aspektien ylläpidettävyys pysyy korkealla tasolla. Tällöin korjaukset ja päivitykset yhteen monitorointitehtävään ei myöskään vaikuta muihin tehtäviin.

Tarkasti määritelty data

Komponenttiin valittu Application Insights-palvelu kerää kaiken metriikkadatan, jota sille lähetetään. Tiedot lähetetään rakenteellisena JSON-datana, joka kuvattiin luvussa 5.5. Tarkan rakenteensa avulla JSON-dataa voidaan myös jäsentää eri tavoin muuhun analyysikäyttöön tarvittaessa. JSON-datan käyttäminen verkkoliikenteen tiedosto-muotona on myös moderni ja paljon käytetty tapa nykypäivänä. Rakenteen pysyessä tarpeeksi pienenä, verkon yli lähetettävän datan määrä on pieni aiheuttaen vain pientä verkkokuormaa. Rakenteisena se on myös helppo tallentaa tietokantaan. JSON rakenteen avaimet voidaan pitää tietokannassa tietosarakkeiden kanssa samannimisinä, jolloin myös jäsentelyn jälkeen datan linkittäminen oikeaan tietokantasarakkeeseen on yksinkertaista.

Käyttötapauksien monitoroinnissa kaikki suorituspolulla mitattavat kohdat voidaan tunnistaa lisämääreillä, joita metriikkaluokan kirjausmetodit tarjoavat. Lisämääreiksi voidaan määrittää käyttötapauksen nimi tai muu tunniste, jonka avulla analyysiä tehdessä kehittäjä voi tarkastella vain tietyn käyttötapauksen suoritusta.

Instrumentoinnin suorituskyky

Komponentti käyttää Application Insights -metriikkarajapintaa tiedon keräämiseen ja lähettämiseen. Application Insights aiheuttaa minimaalista ylimääräistä kuormaa sovellukseen, koska tietoa kerätään ja lähetetään säikeistettynä. Lähetys palvelimelle tapahtuu erikseen pyytäessä (flush-toiminto) tai sopivana valittuna aikana, jolloin käyttäjälle ja järjestelmään aiheutuu mahdollisimman vähän kuormitusta.

Suorituskykyyn vaikuttaa ainoastaan aspekteissa määritelty ylimääräinen metodeissa suoritettava koodi. Koska aspektien ohjeet on luotu siten, että niissä tehtäisiin mahdollisimman vähän ylimääräistä toimintaa, ei suurenkaan aspektimäärän käyttäminen pitäisi vaikuttaa suorituskykyyn huomattavasti.

Ajoitettujen tehtävien luonti on kehittäjän vastuulla ja näin saattaa aiheuttaa suorituskykykustannuksia, mikäli tehtävässä tehdään suurta laskentaa tai käytetään tietokantayhteyksiä ja ladataan dataa tietokannasta.

Tietoturva

Komponentti lähettää kerätyn datan verkon yli ja siksi komponentin pitää olla tietoturvallinen. Komponenttiin valittu Application Insights -rajapinta lähettää kaiken datan salattuna Azure-palvelimille. Application Insights käyttää HTTPS-protokollaa, joka käyttää riittävää salausta metriikkatietojen lähetykseen. Lähetettävissä metriikkatiedoissa anonymisoidaan kaikki salassapidettävä tieto (käyttäjätiedot, asiakas), mikäli sellaista ollaan lähettämässä metriikkatietojen seassa. Näin ollen toteutettua komponenttia voidaan pitää tietoturvallisena tiedon lähettämisen osalta.

Tällä hetkellä Application Insights –palveluun kerätyt tiedot tallennetaan palvelimille, jotka sijaitsevat Yhdysvalloissa. Tästä syystä tietoturvariskit keskittyvät palvelinten sijaintiin ja mahdollisiin palvelinten omiin tietoturva-aukkoihin. Lähetettävän JSON-datan rakenne saattaa aiheuttaa myös mahdollisuuden lähetettävän tiedon kaappaamiseen ja salausavaimen testaamiseen brute force –hyökkäyksen avulla. Jos lähetettävän JSON-datan alkuosio on vakio, voidaan testaamalla etsiä salausavainta, jolla alkuosa saadaan palautettua ja näin saadaan lähetetty paketti palautettua selkokieliseksi. Vaikka hyökkäys onnistuisi, se ei kyseessä olevan metriikkadatan tapauksessa anna hyökkääjälle suurta etua, sillä kerätty data on lähinnä suorituskyvyn mittauksen tuloksia.

Aspektikehyksenä käytetty PostSharp luo suoritettavat aspektit ohjelmaan käännösaikana. Tästä hyötynä saadaan ohjelmiston mukana käännetty aspektikoodi, jota suoritetaan ohjelman sisällä. Toisin kuin metaohjelmassa, jossa yksi ohjelma injektoituu toiseen suoritettavaan ohjelmaan, käännösaikainen aspektipunonta estää ulkopuolisen ohjelman tunkeutumisen järjestelmään aspektipunonnan kautta.

6.1 Ominaisuudet

Ilmainen PostSharp tarjoaa tarvittavat tässä työssä käytettävät aspektikehyksen ominaisuudet. Mikäli ominaisuuksia tarvitaan lisää käytettäväksi, PostSharp-lisenssit joudutaan tarjoamaan kaikille kehittäjille ja käännösservereille, jotka osallistuvat sovelluksen käännösten tekoon. PostSharp-kirjaston lisenssit voivat olla suurikin menoerä pienelle ohjelmistoyritykselle. Tästä syystä työssä toteutettuun komponenttiin ei otettu toistaiseksi käyttöön mitään maksullisen version ominaisuuksia.

Application Insights kuuluu Azure-palveluiden ilmaisiin ominaisuuksiin. Ilmainen versio Application Insights -palvelusta ei kuitenkaan tarjoa kuin 24 tunnin datasäilytyksen, joka suorituskykyanalytiikkaa tehdessä ei ole riittävällä tasolla. Ilmainen versio ei myöskään tarjoa jatkuvaa tiedonvientiä tietokantaan. Maksullisessa versiossa Application Insights:iin on mahdollista lisätä jatkuva datan vienti Azure:n tietokantaan, jossa dataa pystytään säilyttämään suuria määriä metriikkadatan jatkoanalyysiä varten.

6.2 Johtopäätökset

Toteutetun monitorointikomponentin avulla pystytään mittaamaan järjestelmän suorituskykyä niin testiympäristössä kuin myös asiakkaan tuotantoympäristössä. Monitorointia voidaan tehdä metoditasolla järjestelmässä, mutta myös kokonaisia käyttötapauksia ja niiden aiheuttamia tapahtumia voidaan seurata. Monitoroinnin avulla järjestelmästä voidaan löytää mahdollisia pullonkauloja ja muita hitaita operaatioita, joiden optimointi tulee tärkeäksi osaksi järjestelmäkehitystä.

Kun monitorointi otetaan käyttöön useaan järjestelmään, saadaan yhtenevistä toiminnoista suorituskykydataa useiden asiakkaiden järjestelmistä. Luokittelemalla

kerättyä suorituskykydataa voidaan tehdä päätelmiä käytetyn laitteiston tai yhteyksien laadun vaikutuksista järjestelmän suorituskykyyn.

Näkymävaihdosten seurannan perusteella voidaan tarkastella usein käytettyjä näkymiä ja toiminnallisuuksia, joita järjestelmä tarjoaa. Näin saadaan arvokasta tietoa kehityskohteista tai toiminnoista, joiden kehittäminen ja ylläpitäminen suurilla resursseilla on turhaa.

Luvussa 4.1 mainittujen ajoitusalgoritmien ja muiden laajojen suoritusketjujen seurantaa voidaan tehdä komponentin avulla uuden SWD PES -alustaversion myötä. Ajoitustoimintojen laskentayksiköitä voidaan monitoroida erikseen ja tarkastella niiden suorituskykyä. Monitorointikomponentti otetaan käyttöön vuoden 2016 aikana ja näin uusista asiakasratkaisuista, jotka pohjautuvat uuteen alustaversioon saadaan kerättyä suorituskykydataa. Komponentin tarjoamat mahdollisuudet nousevat esille vasta, kun järjestelmästä voidaan kerätä dataa useilta käyttäjiltä.

6.3 Kehitysehdotukset

Aspektikehyksen käyttö suorituskykymonitoroinnissa on loistava tapa saada yhtenevää toiminnallisuutta eriytetyksi järjestelmän alustasta. Erilaisten ei-mitattavien tietojen kerääminen voidaan toteuttaa myös komponenttiin rakennetun aspektikehyksen avulla. Jatkossa suorituskykymonitoroinnin lisäksi laajennetun käyttöanalytiikan kerääminen olisi mahdollista pienin muutoksin järjestelmään. Komponenttiin lisättäisiin käyttöanalytiikkaa palvelevia aspektiohjeita ja näin järjestelmän tarkempi seuraaminen käyttäjätasolla voidaan toteuttaa. Käyttöanalytiikan avulla kehittäjäyritys kykenee arvioimaan tarkemmin käytetyimpiä järjestelmän ominaisuuksia tai mahdollisia ominaisuuksien tuomia ongelmia.

Metriikkaluokan laajennetun rajapinnan avulla voidaan mahdollistaa erilaisten suorituskykymittauksiin tarvittavien toimintojen piilottaminen siten, että rajapinnan käyttäjän ei tarvitse antaa kaikkea tietoa metriikkametodeihin. Metriikkaluokan käyttämän metriikkakirjaston voisi myös abstraktoida siten, että konteksti toimisi abstraktina rajapintana metriikkakirjastolle. Metriikkaluokkaan voitaisiin silloin myös lisätä useita kontekseja tiedon keräämistä varten, jolloin rajoitukset yhden sisäisen kontekstin käyttöön poistuisi. Näin järjestelmästä kerätty metriikkadata olisi mahdollista lähettää useampaan metriikkakontekstiin yhdellä kutsulla. Tämän avulla tietoa voitaisiin lähettää useampaan analytiikkapalveluun tai tallettaa kaikki kerätyt tiedot myös tietokantaan asiakkaan ympäristöön. Usean analytiikkapalvelun käyttäminen monitoroinnin käyttöönottovaiheessa mahdollistaisi useamman palveluiden vertailun. Jos toinen palvelu osoittautuu paremmaksi kokonaisuudeksi monitoroinnin toteutukseen, voidaan osa konteksteista poistaa käytöstä ja keskittyä yhden palvelun käyttöön.

Koska aspekteja käytetään järjestelmässä attribuuttien avulla, komponenttiin olisi tarpeellista jatkokehityksen yhteydessä luoda koostettuja aspektiattribuutteja. Näiden avulla koristeluun vaadittava attribuutti voisi suoraan määritellä useamman monitorointitehtävän suorittamisen sen sijaan, että jokaista tehtävää varten lisättäisiin yksittäinen attribuutti. Tämä siistii koodin ulkonäköä ja helpottaa määrittämään monitorointitehtävien suoritusjärjestyksen, mikäli järjestyksellä on merkitystä.

7. YHTEENVETO

Diplomityön tavoitteena oli toteuttaa ohjelmistokomponentti, jonka avulla ohjelmiston suorituskykymonitorointia voidaan toteuttaa asiakkaan tuotantoympäristössä. Siellä tehtävän monitoroinnin tavoitteena on antaa kehittäjille mahdollisuus seurata järjestelmän suorituskykyä kehitysympäristön ulkopuolella. Näin tehtävä monitorointi antaa paremman kuvan myös kehittäjille järjestelmän toimivuudesta loppukäyttäjän näkökulmasta.

Komponentti toteutettiin osaksi SW-Developmentin Planning Efficiency System (SWD PES) tuotannonsuunnittelujärjestelmää. Komponentin monitorointiominaisuudet ovat käytettävissä kaikissa SWD PES -järjestelmän komponenteissa ja räätälöintiprojekteissa. Komponenttiin toteutettiin metriikkatietojen keräämisestä ja lähettämisestä vastaava metriikkarajapinta. Valmiiseen järjestelmän ohjelmakoodiin ei haluttu tehdä suuria muutoksia, joten tehtävien muutosten minimoimiseksi monitorointimäärittelyt päätettiin toteuttaa aspektiohjelmointia apuna käyttäen. Aspektikehykseen luotiin aspekteja, joiden avulla monitorointikoodi saatiin eriytettyä omaksi kokonaisuudekseen. Aspektipunontaa käyttäen monitorointiominaisuudet lisättiin ohjelmaan käännösaikana ja siksi monitoroitaviin metodeihin ei tarvittu muutoksia.

Komponentti saatiin toteutettua modulaariseksi ja komponentin tärkeimmät osiot, metriikkaluokka ja aspektikokoelma, ovat helposti laajennettavissa tai pienin muutoksin vaihdettavissa toiseen. Komponenttiin valittu Application Insights -metriikkapalvelu osoittautui toimivaksi ja helppokäyttöiseksi ratkaisuksi. Tarvittava tietoturva saadaan hallittua sen tarjoaman salatun verkkoliikenteen avulla. Kaikki palvelun lähettämät tiedot lähetetään HTTPS-protokollaa käyttäen ja mahdolliset asiakasta yksilöivät tunnisteet salataan. Useita yksilöiviä tai asiakkaan tuotantodataan viittaavia tunnisteita ei kuitenkaan ole tarkoitus lähettää metriikkatietojen mukana. Palvelun tarjoaman datan visualisoinnin avulla kehittäjät pystyvät seuraamaan haluamallaan tarkkuudella kerättyä dataa. Luomalla sopivia suodatin- ja ryhmittelykokonaisuuksia datan keräyksen aikana luokitellut tiedot pystytään yksilöimään esimerkiksi käyttötapaus- tai komponentti-kohtaisesti. Tämän avulla tuloksista voidaan tehdä päätelmiä järjestelmän suori-tuskyvystä, mutta löytää myös mahdolliset suorituskyvylliset ongelmakohdat.

Kokonaisuutena toteutettu komponentti on hyvä pohja metriikkadatan keräämiseen tuotannonsuunnittelujärjestelmästä. Komponentti vaatii vielä kehittämistä, mutta tarjoaa perustoiminnot tietojen keräämiseen. Aspektiohjelmoinnin käyttäminen monitoroinnissa luo uusia mahdollisuuksia koko järjestelmän seurantaan mahdollisimman helpoin muutoksin kooditiedostoihin. Attribuuttien käyttö ei sotke koodipohjaan mitään monitorointiin liittyvää koodia, mutta saattaa aiheuttaa hämmennystä kehittäjien

joukossa, kun ei voida olla varmoja, mitä aspektiattribuutin mukana tuomat toiminnot tekevät.

Vuoden 2016 aikana toteutetaan uusi alusta SWD PES -järjestelmään ja suorituskykymonitorointi on tarkoitus ottaa laajasti käyttöön kaikkiin uuden alustan pohjalle tehtyihin asiakasprojekteihin. Kuitenkin ennen monitorointikomponentin käyttöönottoa alustaan ja komponentteihin tehtävää monitorointia on syytä suunnitella. Mitattavat käyttötapaukset ja halutun datan luokittelutunnisteet on tärkeä määritellä ennalta, jotta käyttöönotossa monitoroinnista saatavat tiedot antavat maksimaalisen hyödyn yritykselle ja järjestelmän jatkokehitykselle. Käyttöönottovaiheessa monitorointikomponentin mahdollisuudet ja hyödyt saadaan laajemmin käyttöön ja nähdään, kuinka hyvin komponentti suoriutuu järjestelmän kokonaisvaltaisesta suorituskykymonitoroinnista.

Jatkossa monitorointikomponenttia ja sen aspektikehystä todennäköisesti kehitetään tarjoamaan parempia mahdollisuuksia käyttöanalytiikan tekemiseen. Suorituskykymonitoroinnin ohella tehtävän käyttöanalytiikan avulla yritys luo paremman käsityksen järjestelmän käytöstä ja ohjaa tuotekehitystä oikeille raiteille. Käyttöanalytiikka tuo mukanaan omat haasteensa ja nähtäväksi jää, kuinka hyvin jo toteutettu kehys pystyy näihin haasteisiin vastaamaan.

LÄHTEET

Abdelzad, V., Aliee, F.S. (2011). Aspect-oriented Software Development versus Other Development Methods, Journal of Theoretical and Applied Information Technology, 31(2), pp.147–152.

Aksit, M., Tekinerdogan, B., Bergmans, L. (1996). Achieving Adaptability through Separation and Composition of Concerns, Special Issues in ObjectOriented Programming, pp.12–23.

Bergmans, L., Lopes, C.V. (1999). Aspect-Oriented Programming, ECOOP'99 Workshops, 2, pp.288–313.

Bryan, P.C. & Zyp, K. (2012). JSON Reference, Internet Engineering Task Force (IETF) Draft. Available (accessed 25.9.2015): http://tools.ietf.org/html/draft-pbryan-zyp-json-ref-03.

Cole, O. (2004). Aprobe : A Non-Intrusive Framework for Software Instrumentation. Available (accessed 19.9.20152015): http://www.ocsystems.com/pdf/ AprobeTechnologyOverview.pdf.

Ferguson, J., Patterson, B., Beres, J., Boutquin, P., Gupta, M. (2002). C # Bible, Wiley Publishing, 830p.

Freeman, A. (2010). .NET 4 Parallel Programming in C#, Apress, 328p. Available (accessed 20.10.2015): http://www.apress.com/9781430229674.

Hodges, N. (2014). Coding in Delphi, Nepeta Enterprises, 242p.

Huang, J.C. (1978). Program Instrumentation and Software Testing, Computer, 11(4), pp.25–32.

Jacobson, I., Ng, P.-W. (2005). Aspect-Oriented Software Development with Use Cases, Addison-Wesley, 464p.

Joyce, G.S. (1967). Classical Heisenberg Model. Physical Review, 155(2), 478p. Available (accessed 22.10.2015): http://link.aps.org/doi/10.1103/PhysRev.155.478\ nhttp://prola.aps.org/abstract/PR/v155/i2/p478_1

Karamath, E., Desharnais, J.-M. (2010). Cyclomatic Complexity in Software Maintenance, Framework, pp.1–8.

Kellens, A., Mens, K., Brichau, J., Gybels, K. (2006). Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts, ECOOP 2006 Object-Oriented Programming, pp.501–525.

Khanna, G., Beaty, K., Kar, G., Kochut, A. (2006). Application Performance Management in Virtualized Server Environments, IEEEIFIP Network Operations and Management Symposium NOMS 2006, 20(D), pp.373–381.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J. (1997). Aspect-oriented Programming. Computer Science, 1241/1997, pp.220–242.

Kiczales, G., Mezini, M. (2005). Aspect-oriented Programming and Modular Reasoning, 27th International Conference on Software Engineering, pp.49–58.

Koliai, S., Zuckerman, S., Oseret, E., Ivascot, M., Moseley, T., Quang, D., Jalby, W. (2010). A Balanced Approach to Application Performance Tuning. Lecture Notes in Computer Science, 5898, pp.111–125. Available (accessed 28.9.2015): http://link.springer.com.ezproxy.auckland.ac.nz/chapter/10.1007/978-3-642-13374-9_8.

Lincke, R., Lundberg, J., Löwe, W. (2008). Comparing Software Metrics Tools. In Proceedings of the 2008 International Symposium on Software Testing and Analysis. ISSTA ’08. New York, NY, USA: ACM, pp. 131–142.

Lippman, S.B. (2002). Essential C++, Addison-Wesley, 416p.

Margarintescu, I. (2015). Metrics .Net wiki. Available at: https://github.com/etishor/Metrics.NET/wiki [Accessed September 1, 2015].

Mazurov, A., Couturier, B. (2012). Advanced Modular Software Performance Monitoring. Journal of Physics: Conference Series, 396(5), pp.1-14.

Microsoft Azure (2010). Windows Azure Virtual Network | Windows Azure Platform, Microsoft. Available (accessed: 15.10.2015): http://www.microsoft.com/ windowsazure/virtualnetwork/ 18/04/11.

Murphy, G., Schwanninger, C. (2006). Aspect-oriented Programming, IEEE Software, 23(1), pp.20–23.

Ndem, G.C., Tahir, A., Ulrich, A., Goetz, H. (2011). Test data to reduce the complexity of unit test automation, Proceedings of the 6th International Workshop on Automation of Software Test, pp.105–106.

Peltola, J. (2015). Tuotannonohjausjärjestelmän automaatiorajapinnan suunnittelu, diplomityö, Tampereen Teknillinen Yliopisto, Tietotekniikan laitos, Tampere. 58 s.

PostSharp (2015). PostSharp Documentation. Available (accessed 14.8.2015): doc.postsharp.net.

Ridgeway, M. (2008). Mastering Microsoft Visual Basic 2008, Wiley Publishing, 1152p.

Rosenberg, L., Hyatt, L. (1997). Software Quality Metrics for Object-oriented Environments, Crosstalk Journal, April, 10(4), pp.1–6.

Sabharwal, C.L. (1998). Java, Java, Java, IEEE Potentials, 17(3), pp.33–37.

Smith, C.U., Williams, L.G. (2001). Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Addison-Wesley, 510p.

SW-Development (2015). SW-Development verkkosivut. Available (accessed 2.10.2015): www.swd.fi.

Thai, T., Lam, H.Q. (2001). NET Framework Essentials, O'Reilly Media, 384p.

Thompson, O. (2006). Does MES contribute to ERP success? Food Engineering, (October), p.119.

Wills, A.C. (2015). Microsoft, Application Insights Documentation. Available (accessed 28.8.2015): https://azure.microsoft.com/en-us/documentation/services/application-insights/.

Woodside, M., Franks, G., Petriu, D.C. (2007). The Future of Software Performance Engineering, Future of Software Engineering (FOSE '07), pp.171–187. Available (accessed 3.9.2015): http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4221619.